SIGMOD 2015, Melbourne, Victoria, Australia

Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou
Columbia University

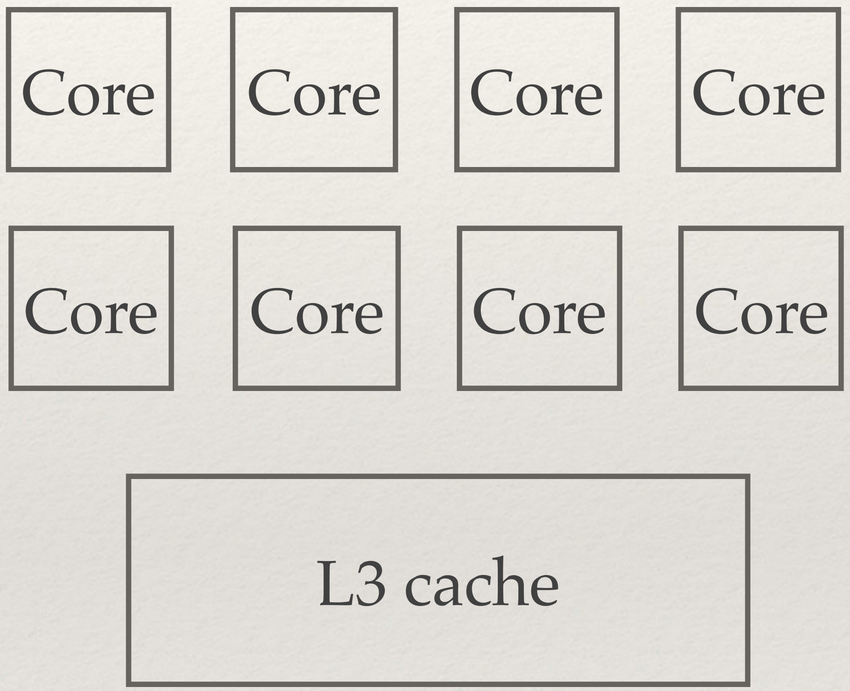
Arun Raghavan
Oracle Labs

Kenneth A. Ross
Columbia University

Latest Hardware Designs

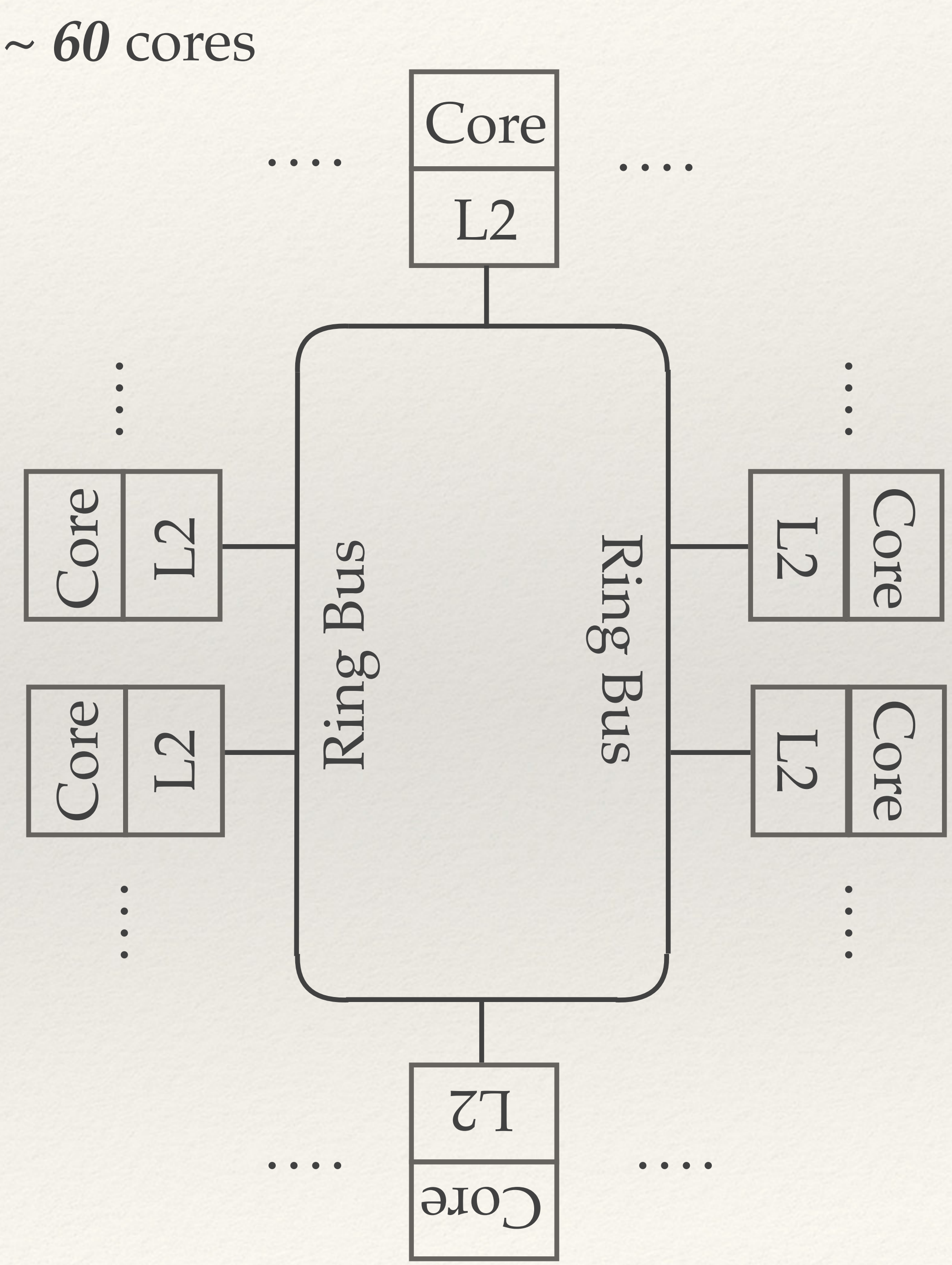
- Mainstream multi-core CPUs
  - Use ***complex*** cores (e.g. Intel Haswell)
    - Massively ***superscalar***
    - Aggressively ***out-of-order***
  - Pack ***few*** cores per chip
    - High ***power*** & ***area*** per core

2 – 18 cores



Latest Hardware Designs

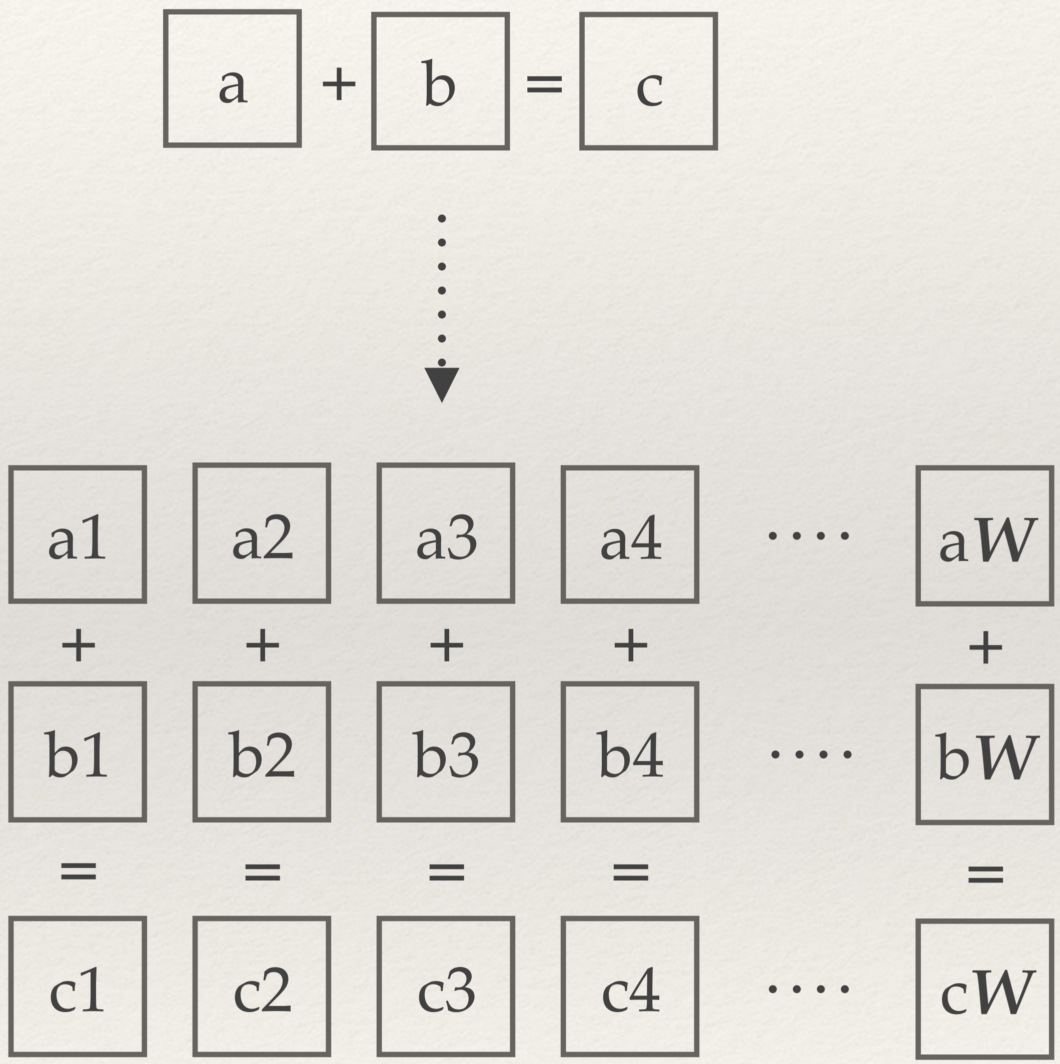
- Mainstream multi-core CPUs
  - Use ***complex*** cores (e.g. Intel Haswell)
    - Massively ***superscalar***
    - Aggressively ***out-of-order***
  - Pack ***few*** cores per chip
    - High ***power*** & ***area*** per core
- Many Integrated Cores (MIC)
  - Use ***simple*** cores (e.g. Intel P54C)
    - ***In-order*** & ***non-superscalar*** (for SIMD)
  - ***Augment*** SIMD to bridge the gap
    - Increase SIMD ***register size***
    - More ***advanced*** SIMD instructions
  - Pack ***many*** cores per chip
    - Low ***power*** & ***area*** per core



SIMD & Databases

- Automatic vectorization
  - Works for ***simple*** loops only
    - ***Rare*** in database operators

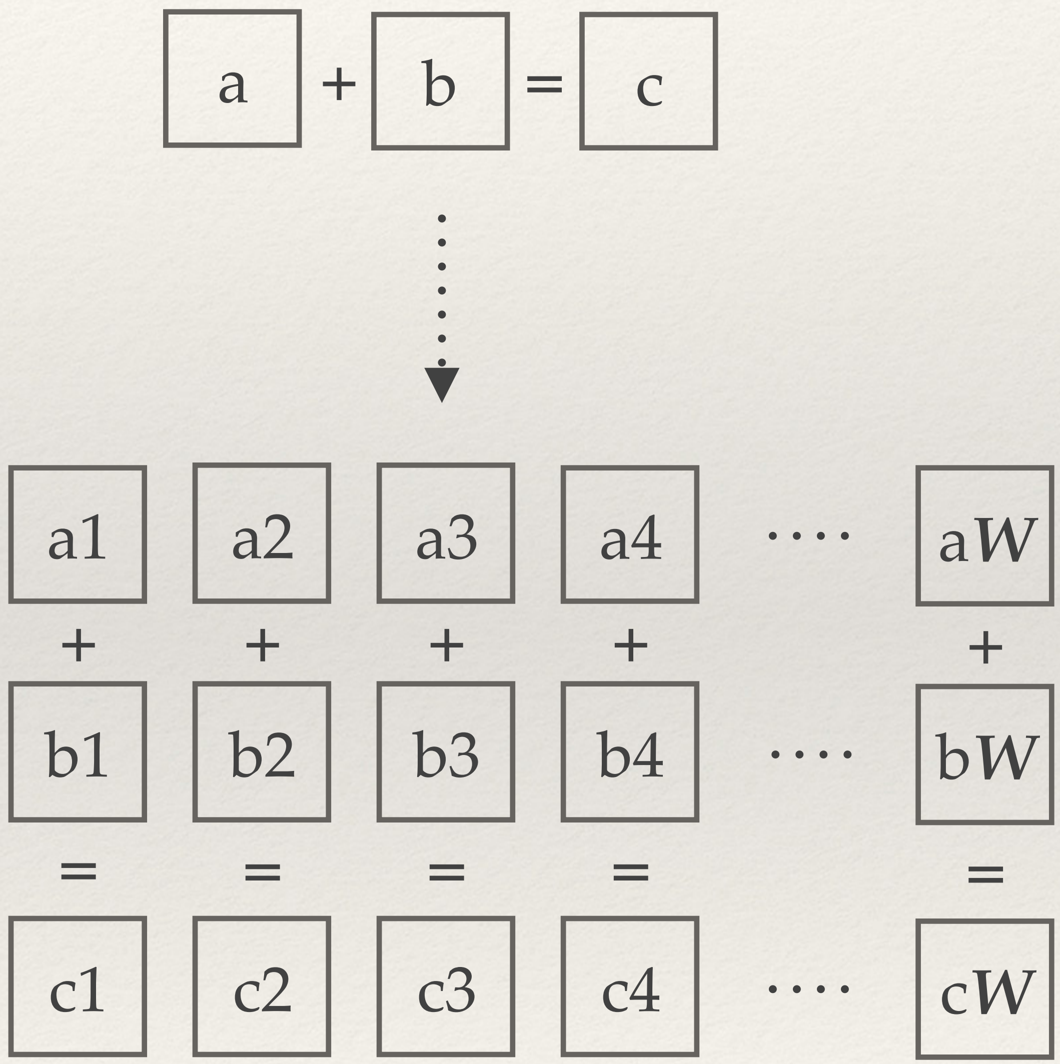
*what is **SIMD** ?*



SIMD & Databases

- Automatic vectorization
  - Works for ***simple*** loops only
    - ***Rare*** in database operators
- Manual vectorization
  - ***Linear*** access operators
    - Predicate evaluation
    - Compression
  - ***Ad-hoc*** vectorization
    - Sorting (e.g. merge-sort, comb-sort, …)
    - Merging (e.g. 2-way, bitonic, …)
  - Generic vectorization
    - ***Multi-way*** trees
    - ***Bucketized*** hash tables

*what is **SIMD** ?*



Contributions & Outline

- ***Full*** vectorization
  - From $O(f(n))$ scalar to $O(f(n)/W)$ vector operations
    - ***Random*** accesses excluded
  - ***Principles*** for good (***efficient***) vectorization
    - Reuse fundamental ***operations*** across multiple vectorizations
    - Favor ***vertical vectorization*** by processing different input data per lane
    - Maximize lane ***utilization*** by executing different things per lane subset

Contributions & Outline

- ***Full*** vectorization
  - From $O(f(n))$ scalar to $O(f(n)/W)$ vector operations
    - ***Random*** accesses excluded
  - ***Principles*** for good (***efficient***) vectorization
    - Reuse fundamental ***operations*** across multiple vectorizations
    - Favor ***vertical vectorization*** by processing different input data per lane
    - Maximize lane ***utilization*** by executing different things per lane subset

- Vectorize ***basic*** operators
  - Selection scans
  - Hash tables
  - Partitioning

Contributions & Outline

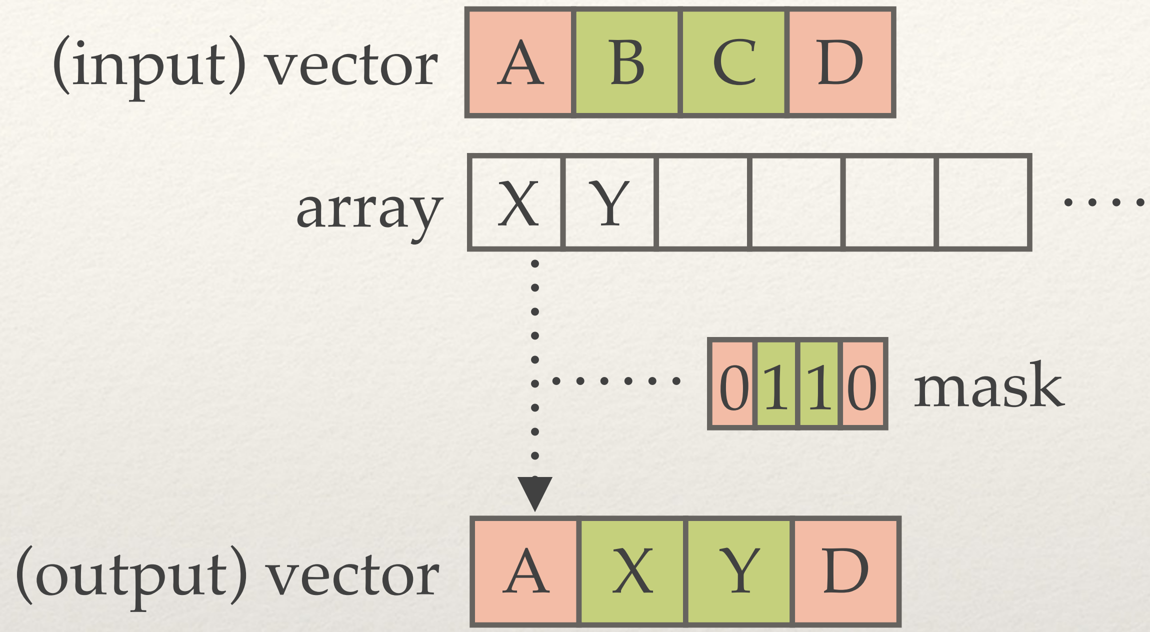
- ***Full*** vectorization
  - From $O(f(n))$ scalar to $O(f(n)/W)$ vector operations
    - ***Random*** accesses excluded
  - ***Principles*** for good (***efficient***) vectorization
    - Reuse fundamental ***operations*** across multiple vectorizations
    - Favor ***vertical vectorization*** by processing different input data per lane
    - Maximize lane ***utilization*** by executing different things per lane subset
- Vectorize ***basic*** operators & build ***advanced*** operators (in memory)
  - Selection scans
  - Hash tables
  - Partitioning
  - Sorting
  - Joins

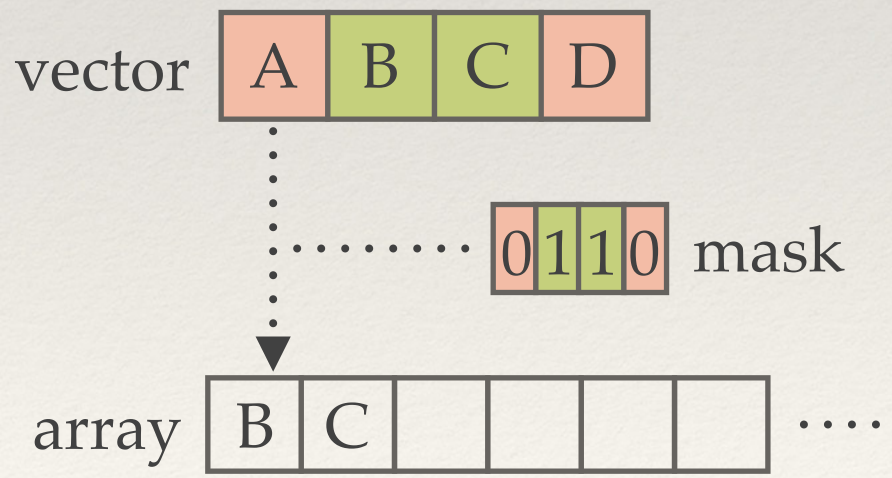
Contributions & Outline

- ***Full*** vectorization
  - From $O(f(n))$ scalar to $O(f(n)/W)$ vector operations
    - ***Random*** accesses excluded
  - ***Principles*** for good (***efficient***) vectorization
    - Reuse fundamental ***operations*** across multiple vectorizations
    - Favor ***vertical vectorization*** by processing different input data per lane
    - Maximize lane ***utilization*** by executing different things per lane subset
- Vectorize ***basic*** operators & build ***advanced*** operators (in memory)
  - Selection scans
  - Hash tables
  - Partitioning
  - Sorting
  - Joins
- Show impact of ***good*** vectorization
  - On ***software*** (database system) & ***hardware*** design

Fundamental Vector Operations

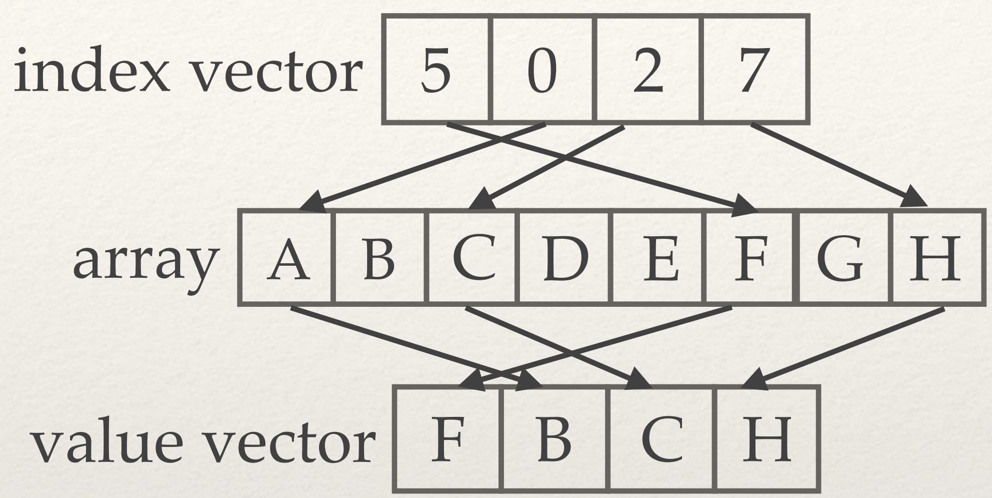
- Selective load
- Selective store

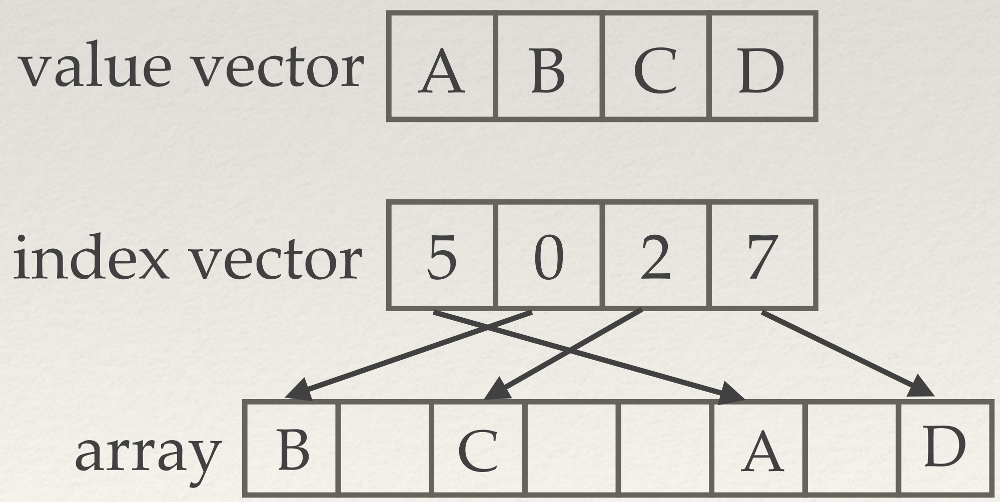




Fundamental Vector Operations

- Selective load
- Selective store
- (Selective) gather
- (Selective) scatter



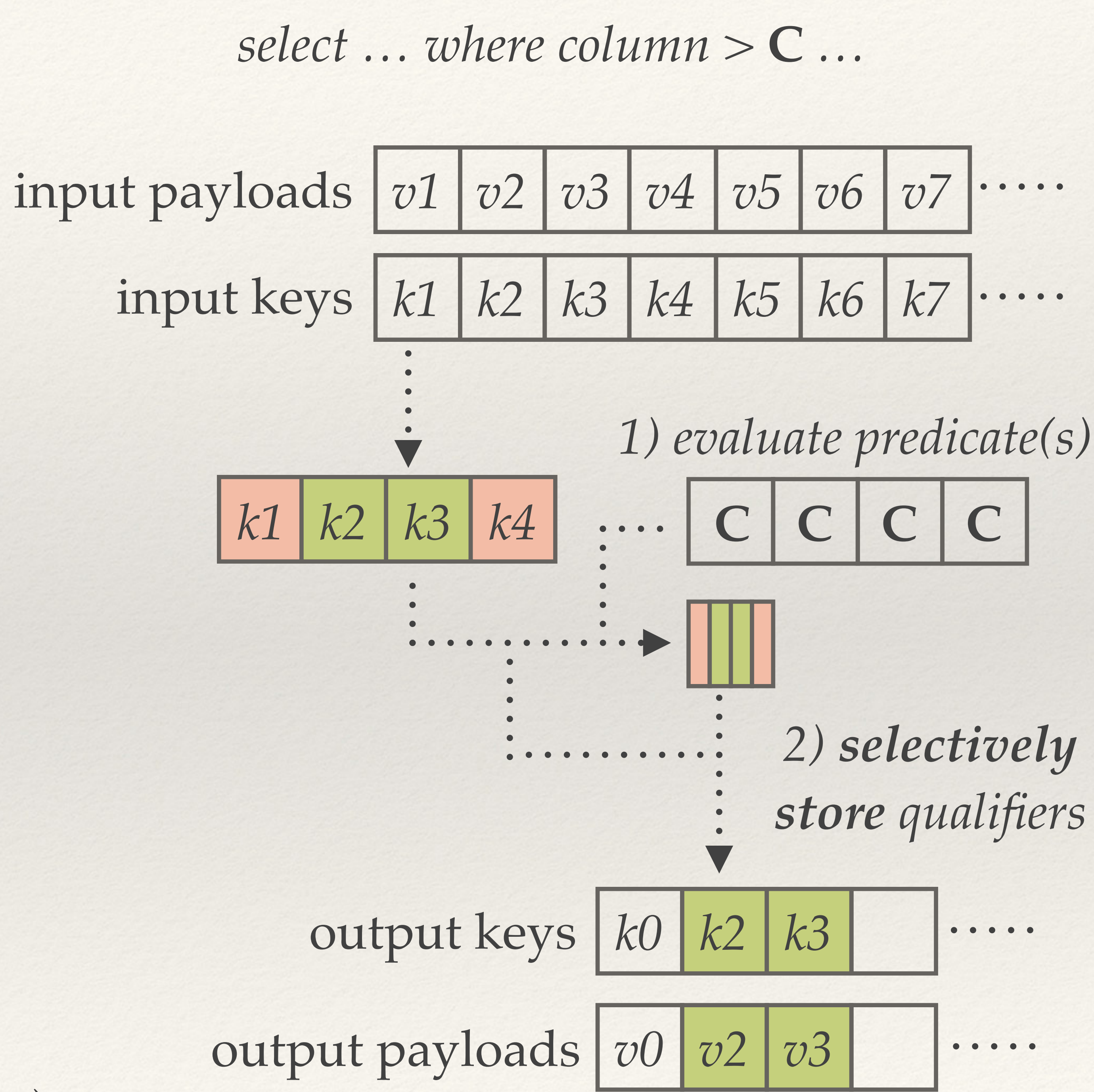


Selection Scans

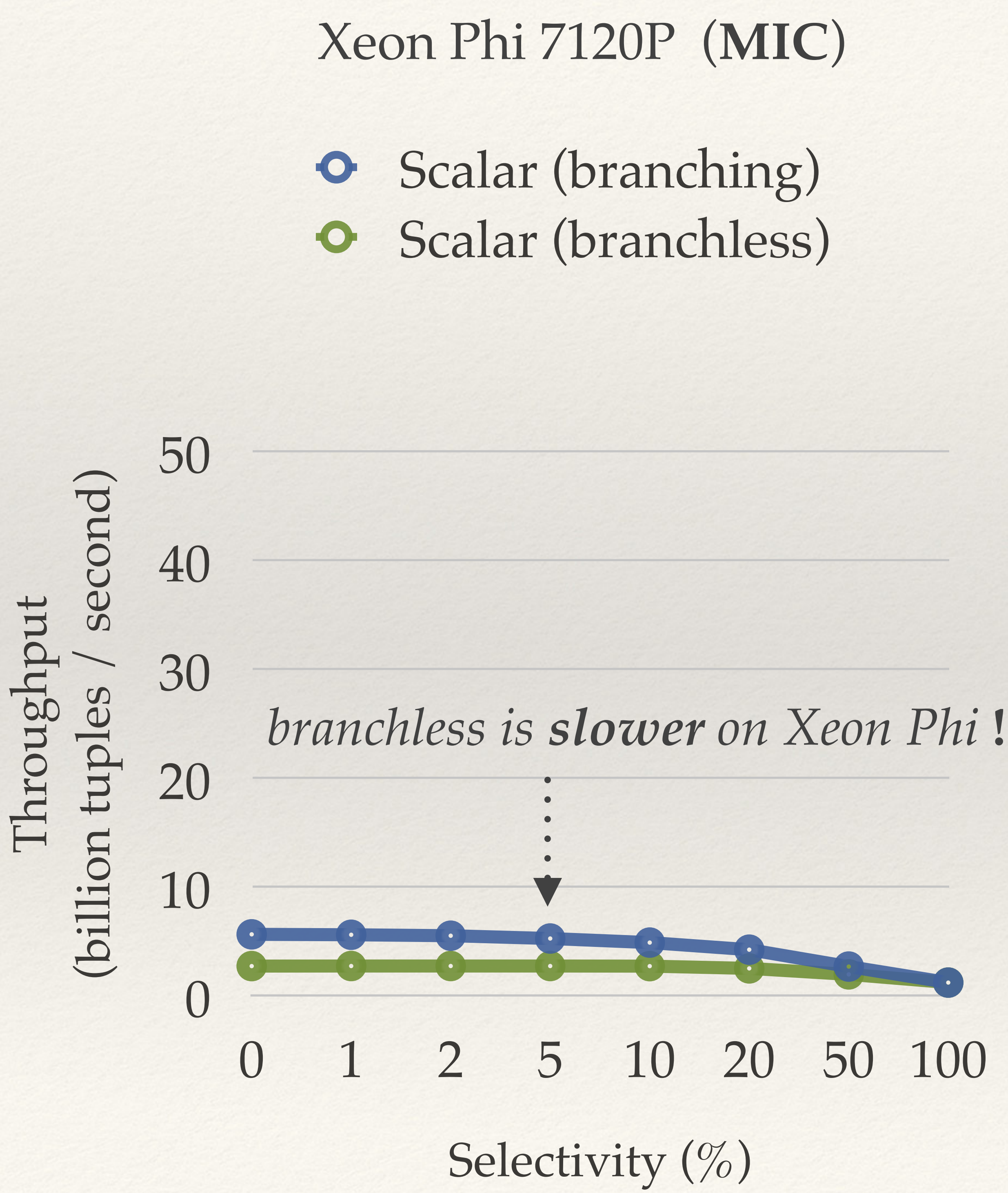
- Scalar
  - Branching
    - ***Branch*** to store qualifiers
    - Affected by ***branch misses***
  - Branchless
    - ***Eliminate*** branching
    - Use conditional ***flags***

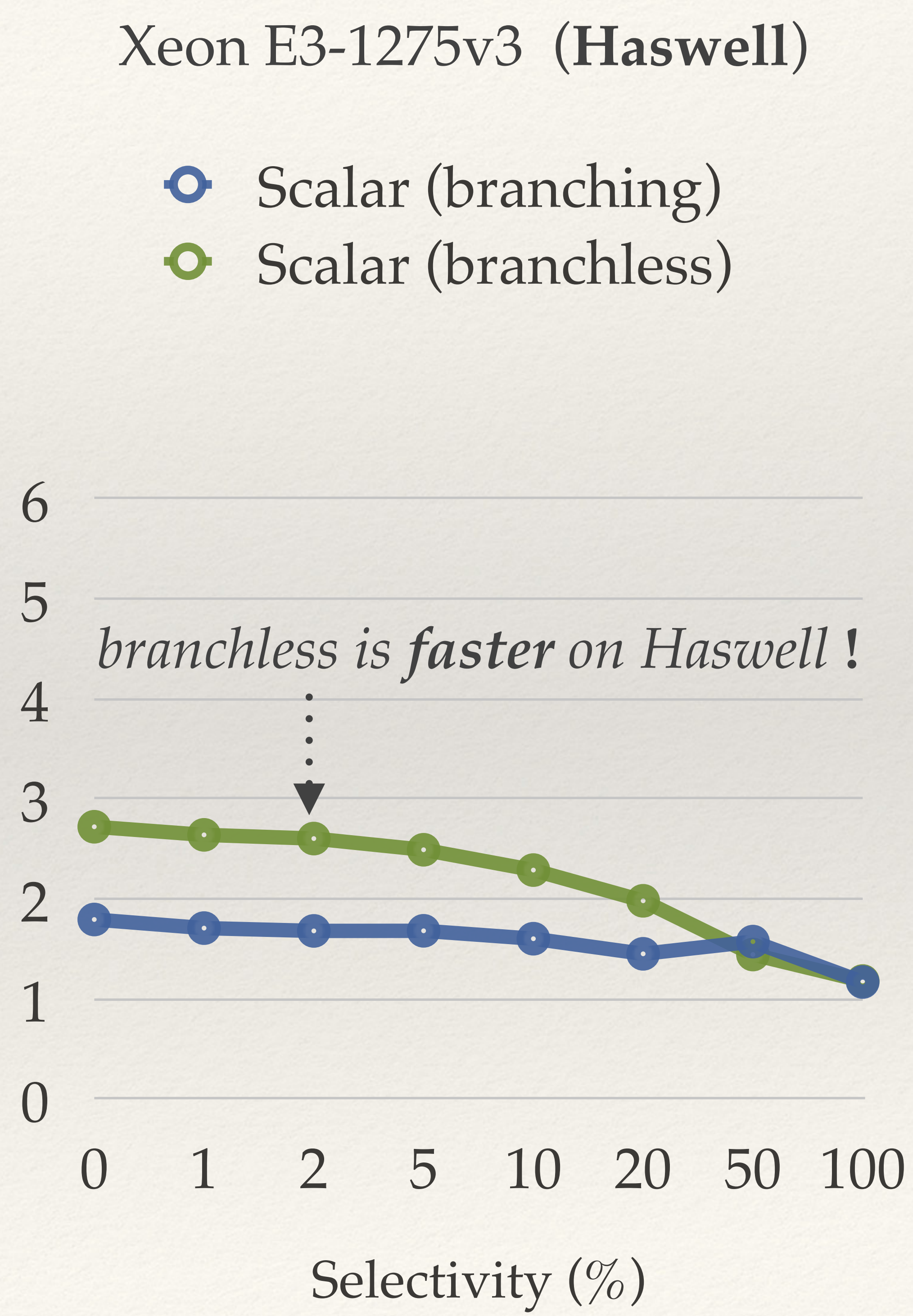
Selection Scans

- Scalar
  - Branching
    - ***Branch*** to store qualifiers
    - Affected by ***branch misses***
  - Branchless
    - ***Eliminate*** branching
    - Use conditional ***flags***
- Vectorized
  - Simple
    - Evaluate predicates (in SIMD)
    - ***Selectively store*** qualifiers
    - "Early" materialized
  - Advanced
    - "Late" materialized (in the paper …)

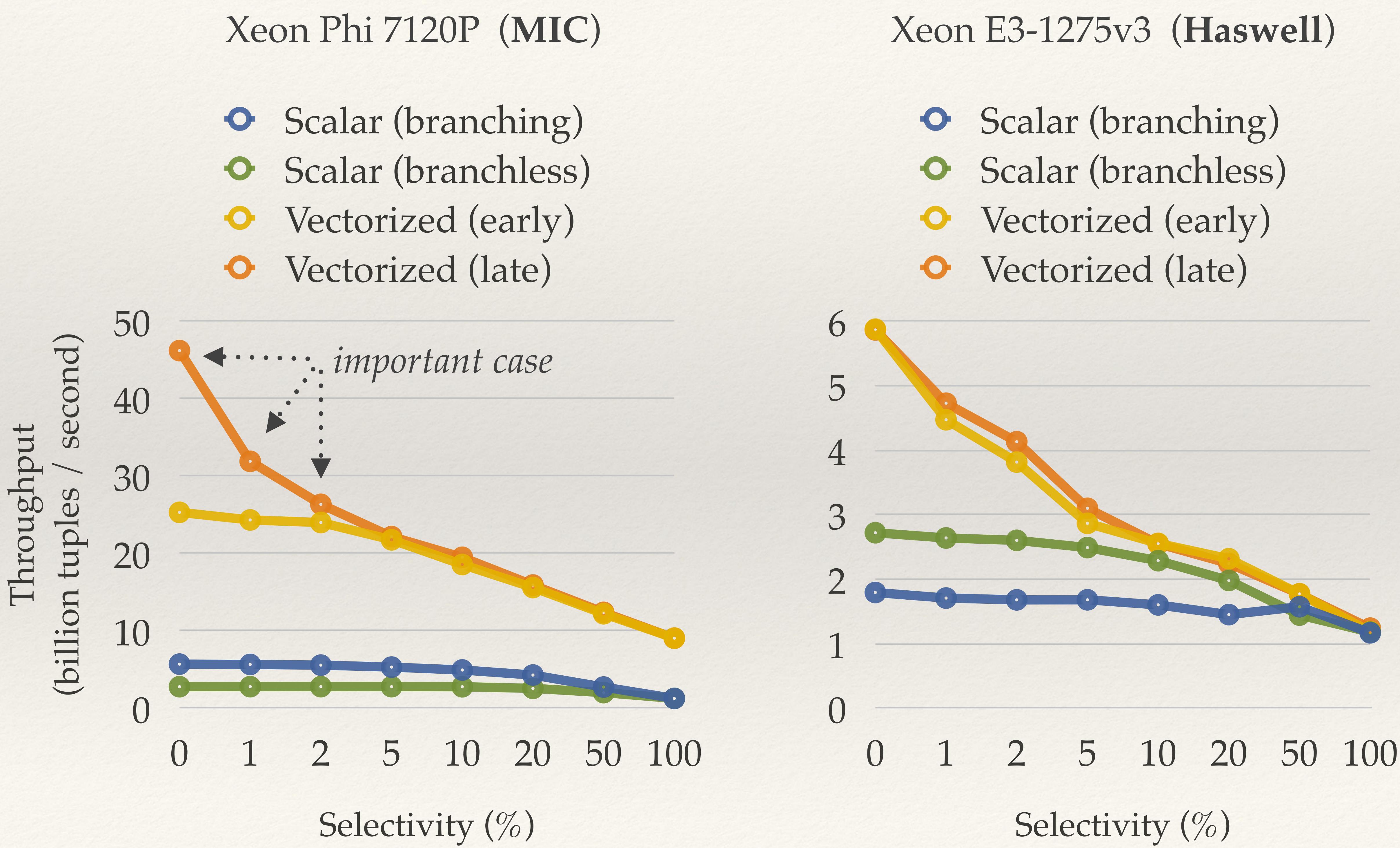


Selection Scans



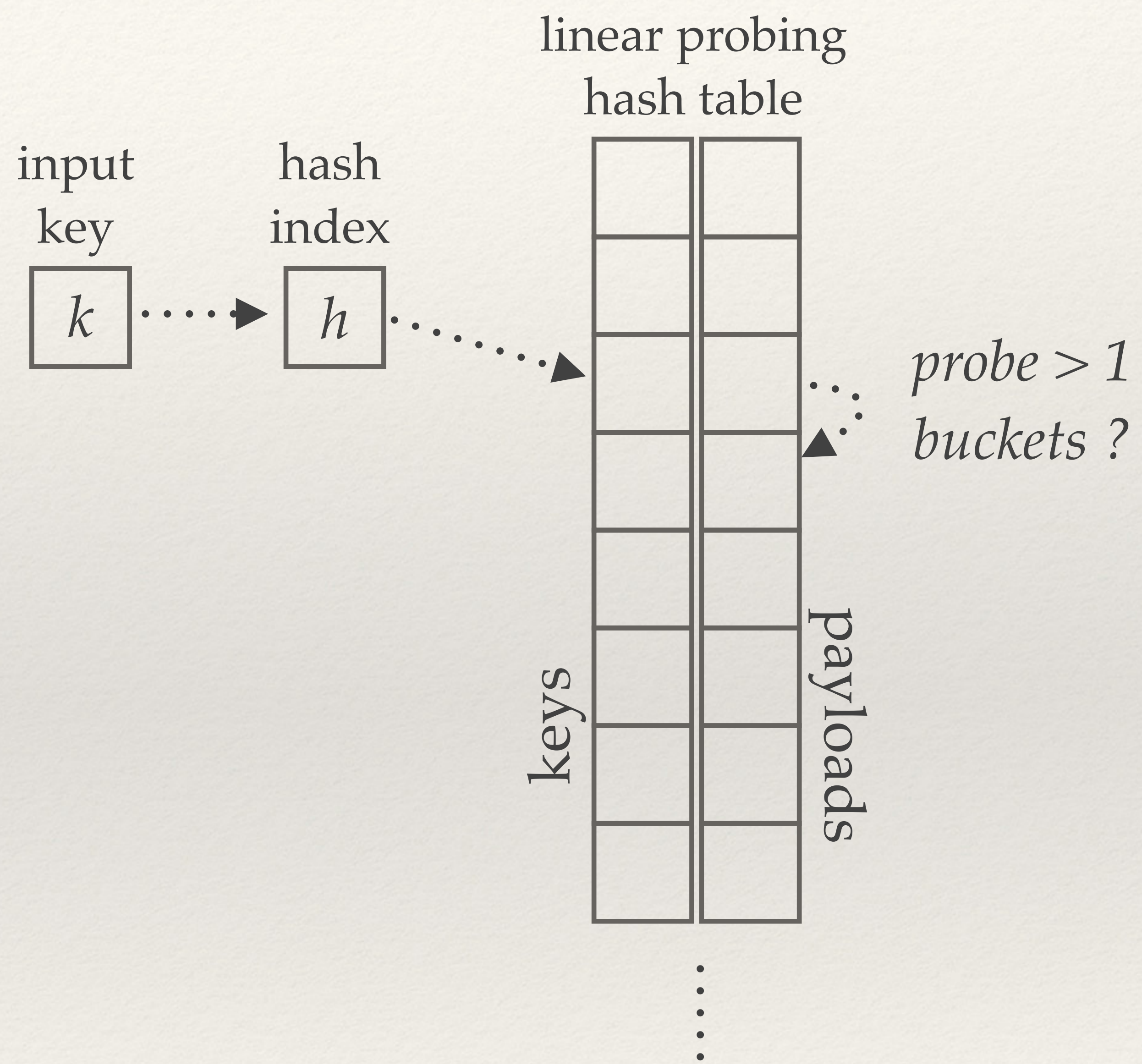


Selection Scans



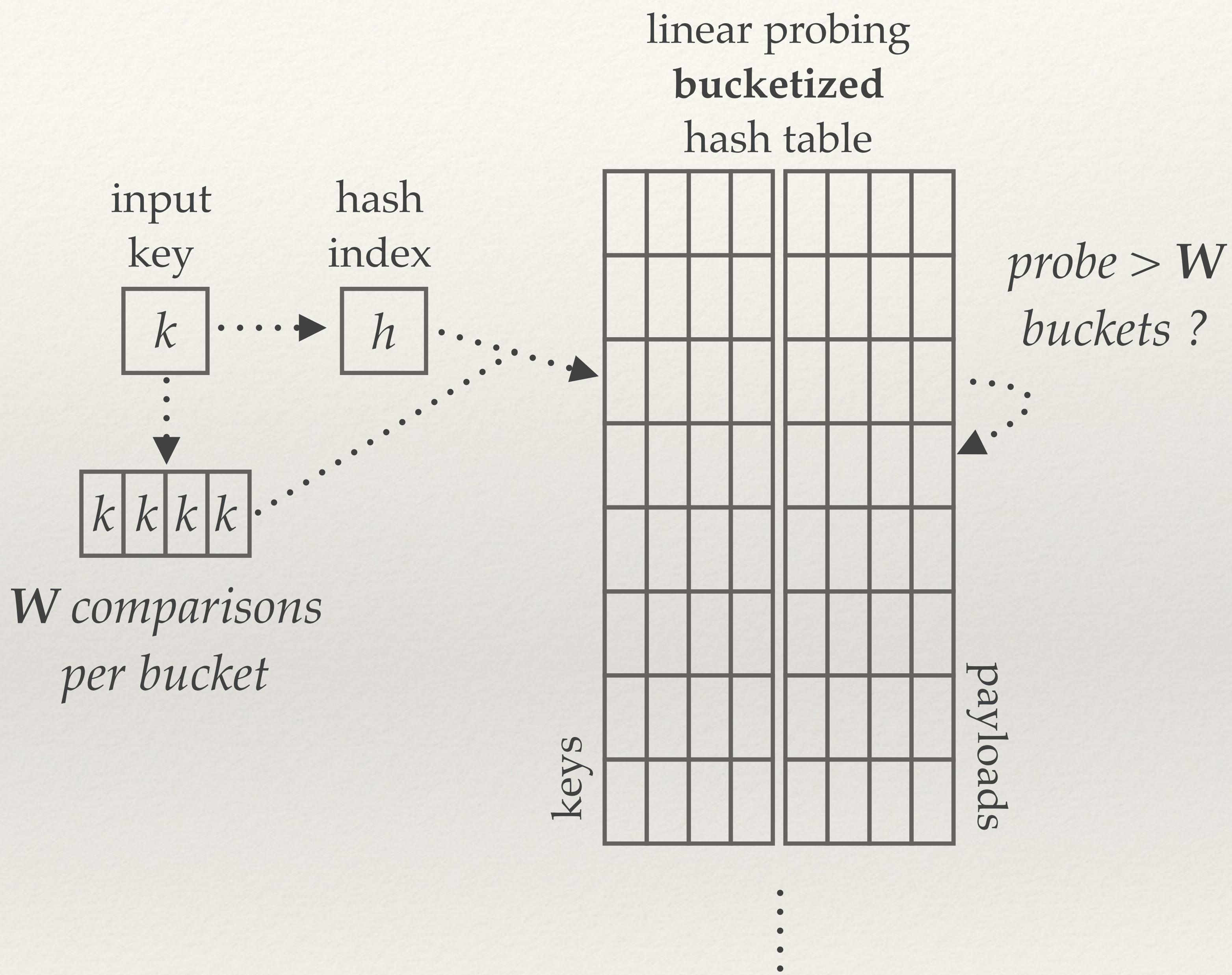
Hash Tables

- Scalar hash tables
  - Branching or branchless
    - 1 ***input*** key at a time
    - 1 ***table*** key at a time



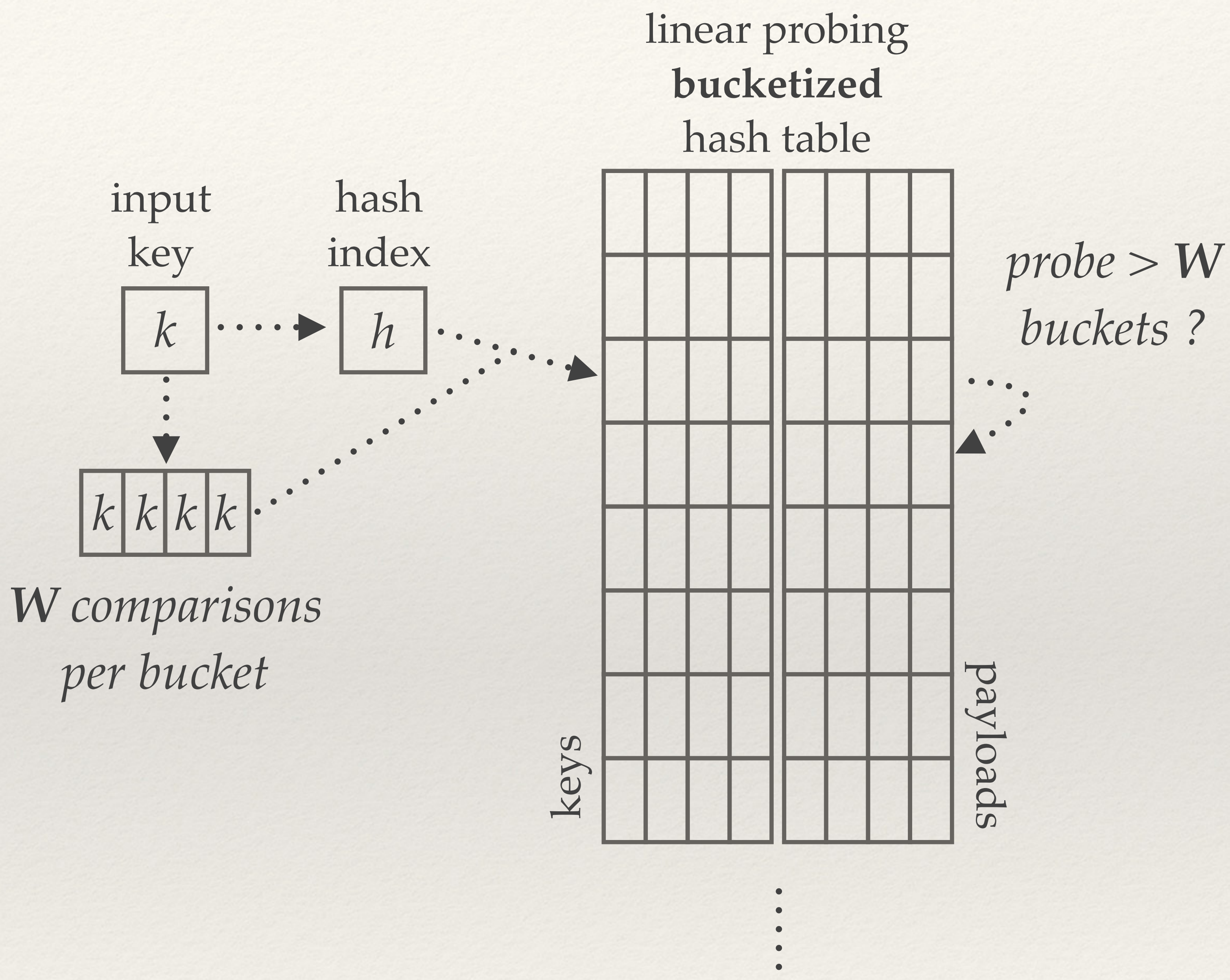
Hash Tables

- Scalar hash tables
  - Branching or branchless
    - 1 ***input*** key at a time
    - 1 ***table*** key at a time
- Vectorized hash tables
  - ***Horizontal*** vectorization
    - Proposed on ***previous work***
    - 1 ***input*** key at a time
    - W ***table*** keys per input key
    - ***Load*** bucket with ***W*** keys



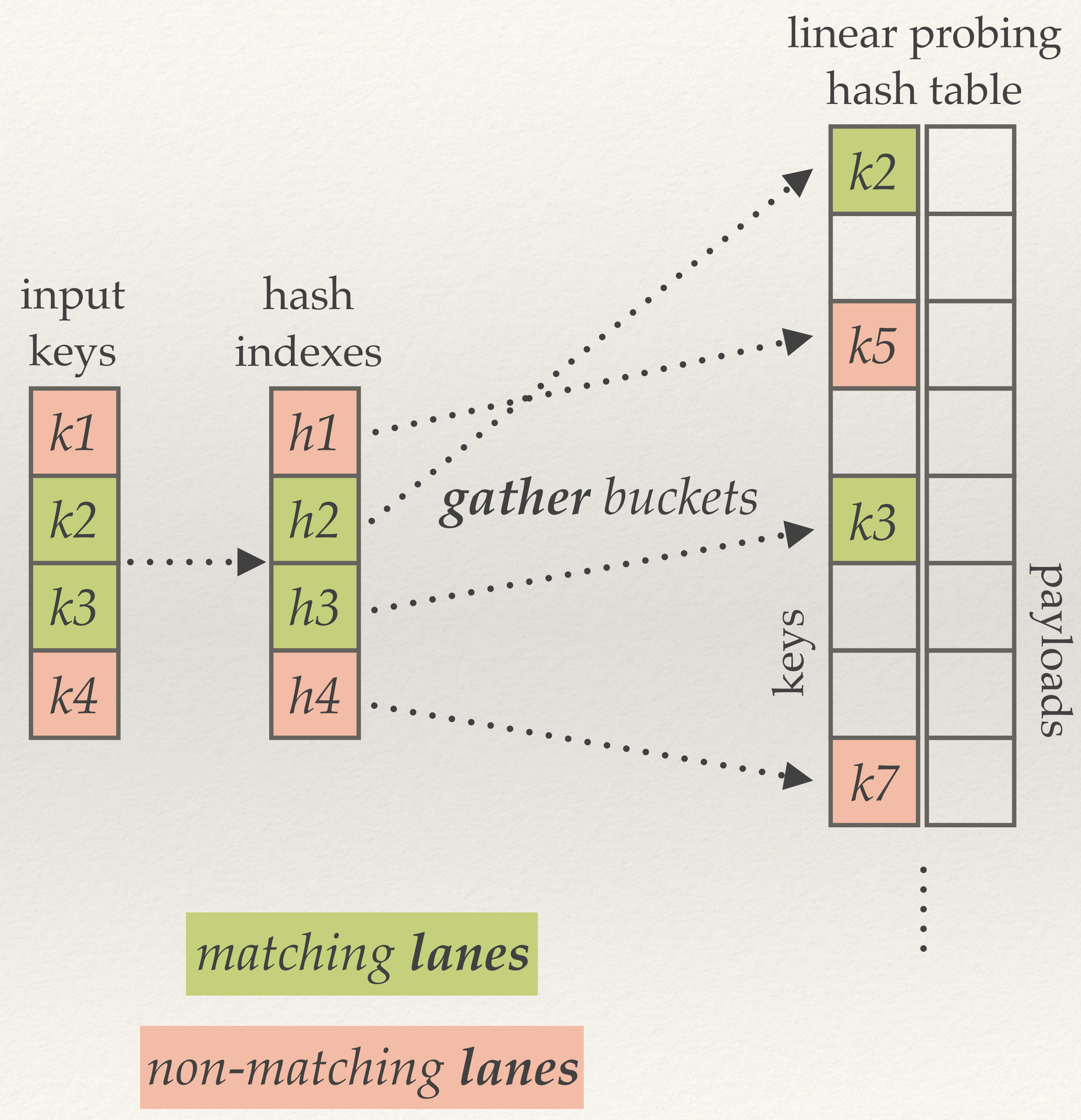
Hash Tables

- Scalar hash tables
  - Branching or branchless
    - 1 ***input*** key at a time
    - 1 ***table*** key at a time
- Vectorized hash tables
  - ***Horizontal*** vectorization
    - Proposed on ***previous work***
    - 1 ***input*** key at a time
    - W ***table*** keys per input key
    - ***Load*** bucket with ***W*** keys
  - However …
    - ***W*** are too many comparisons
    - No advantage of larger SIMD



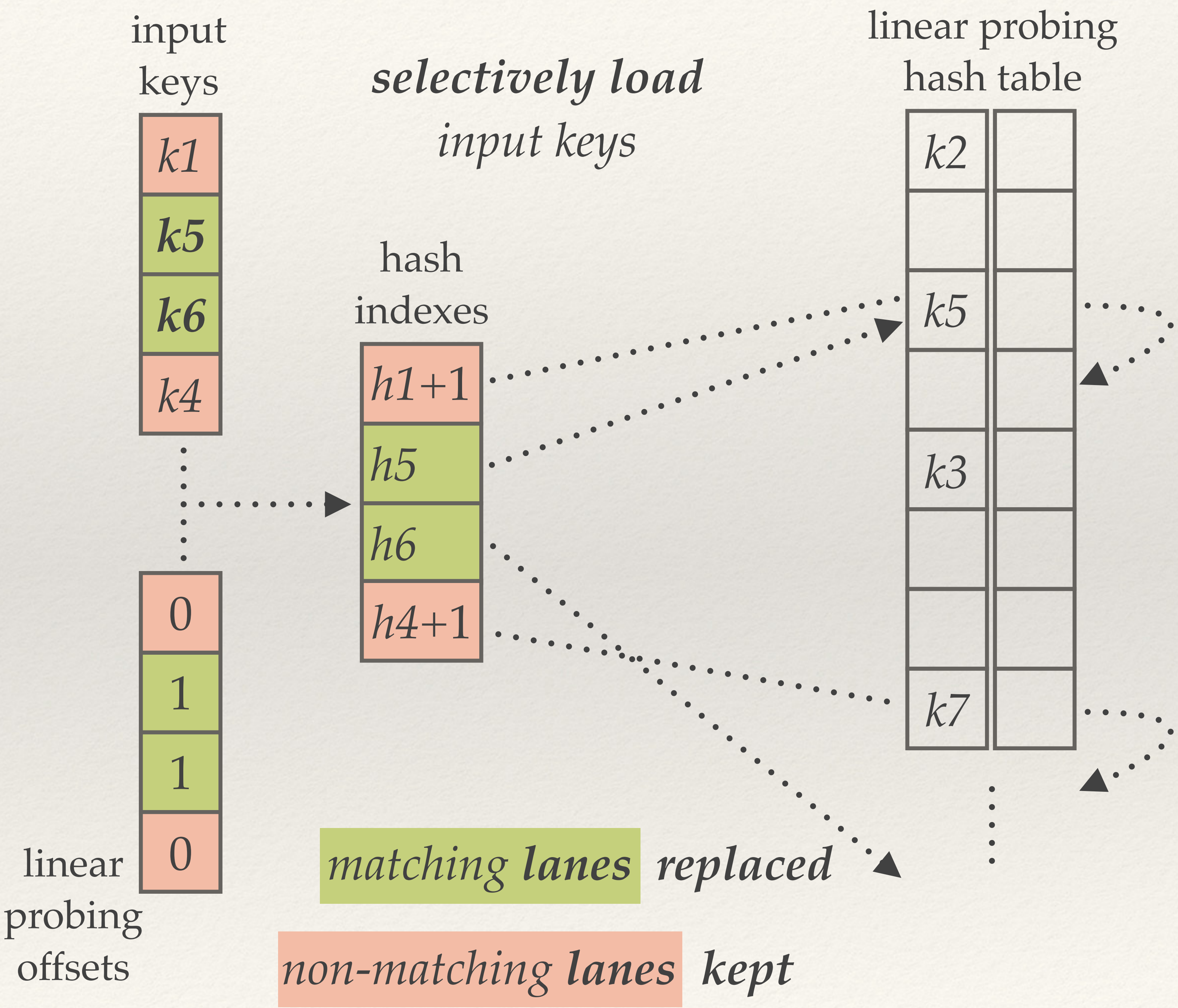
Hash Tables

- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes



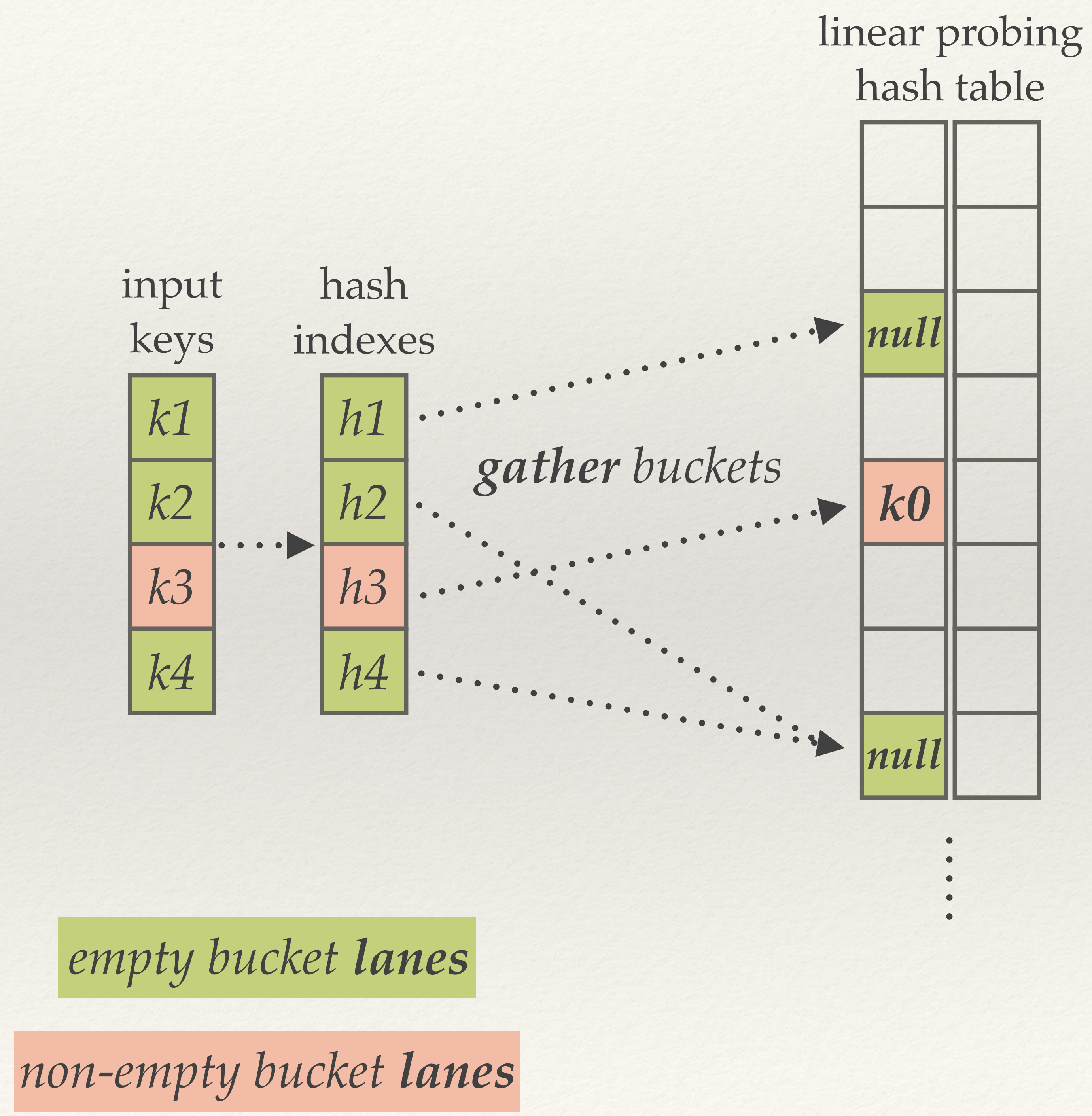
Hash Tables

- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes
    - ***Replace*** finished lanes
    - ***Keep*** unfinished lanes



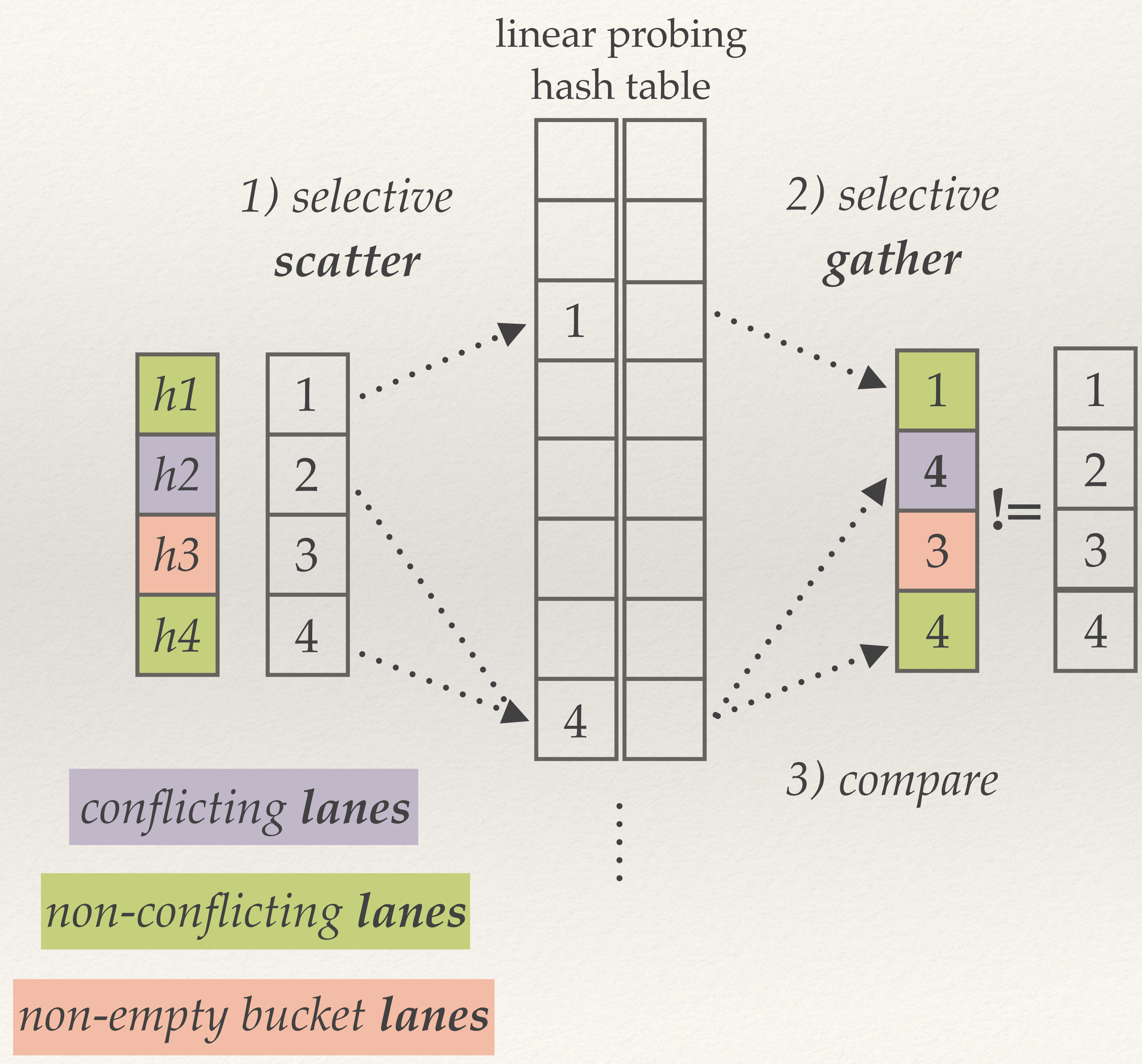
Hash Tables

- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes
    - ***Replace*** finished lanes
    - ***Keep*** unfinished lanes
  - Building *(linear probing)*
    - ***Keep*** non-empty lanes



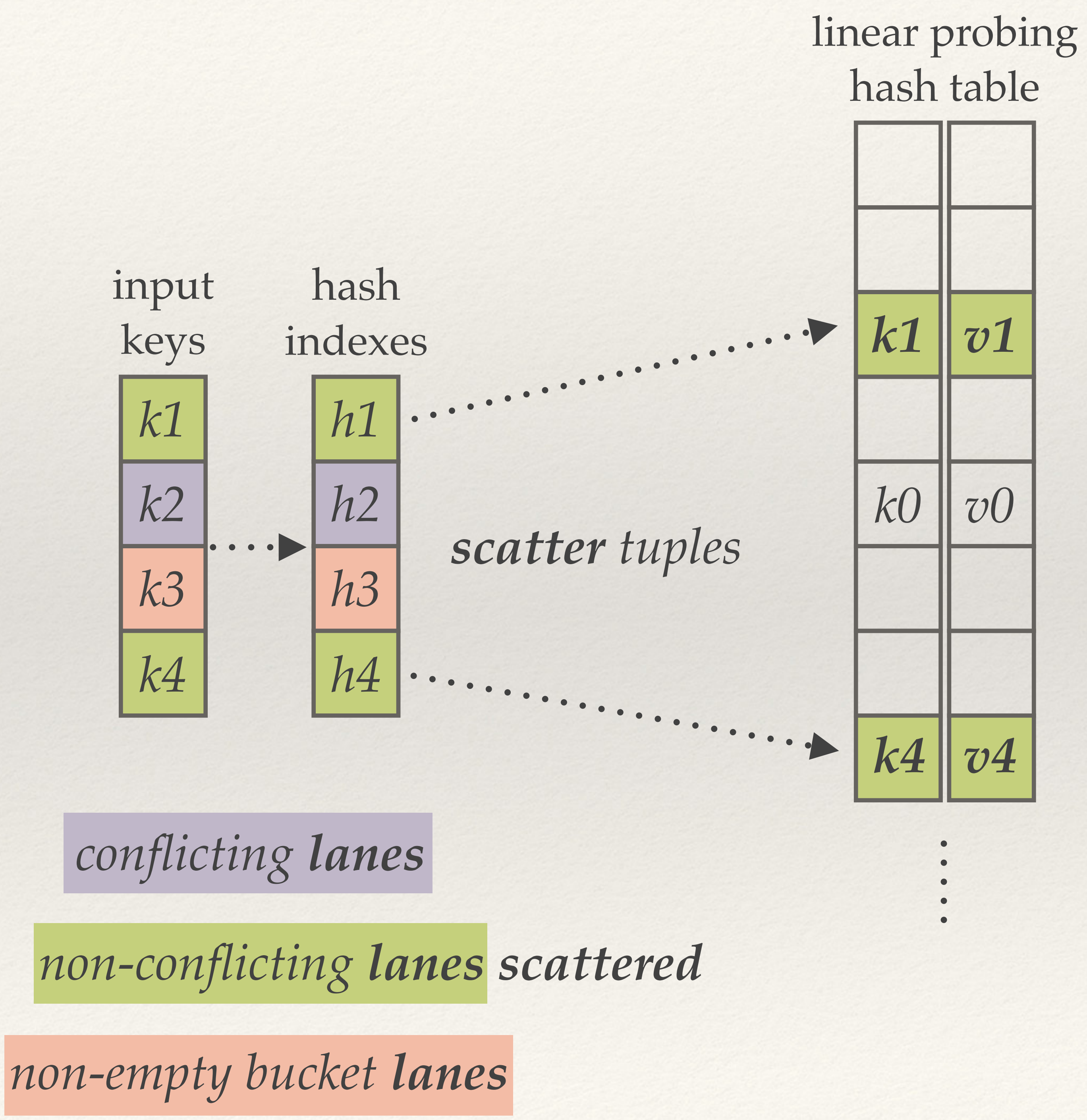
Hash Tables

- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes
    - ***Replace*** finished lanes
    - ***Keep*** unfinished lanes
  - Building *(linear probing)*
    - ***Keep*** non-empty lanes
    - ***Detect*** scatter ***conflicts***



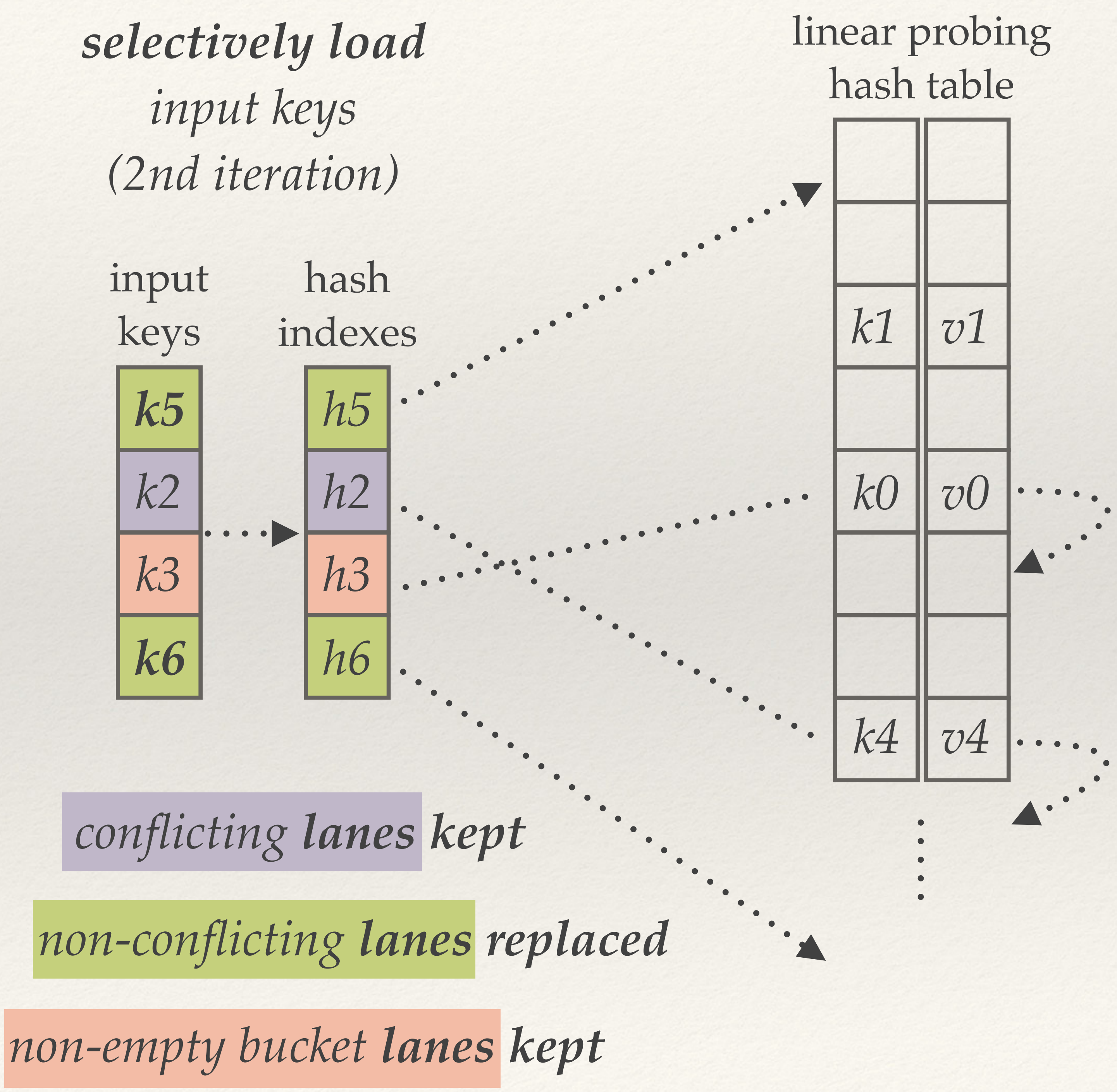
Hash Tables

- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes
    - ***Replace*** finished lanes
    - ***Keep*** unfinished lanes
  - Building *(linear probing)*
    - ***Keep*** non-empty lanes
    - ***Detect*** scatter ***conflicts***
    - ***Scatter*** empty lanes

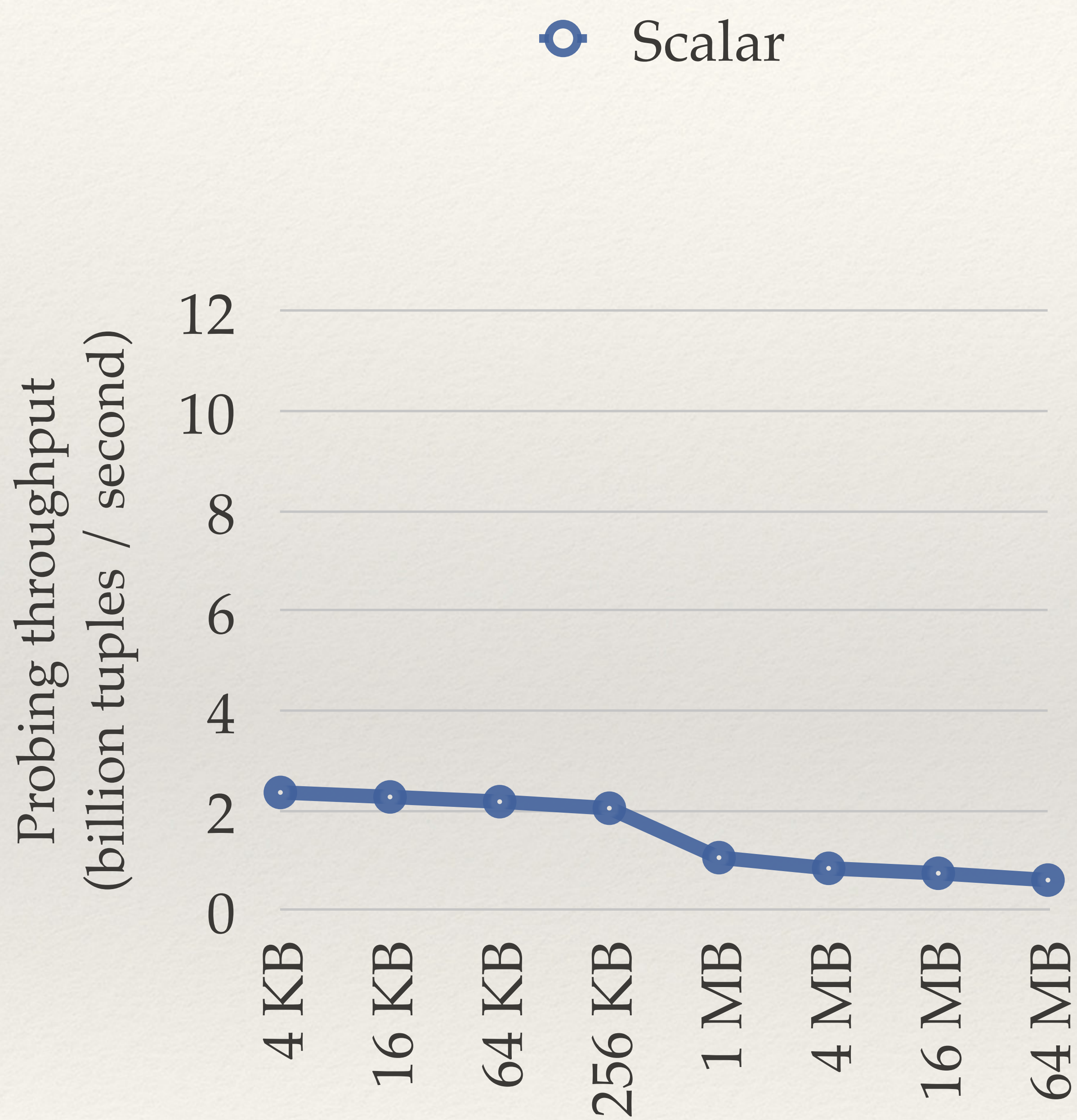


Hash Tables

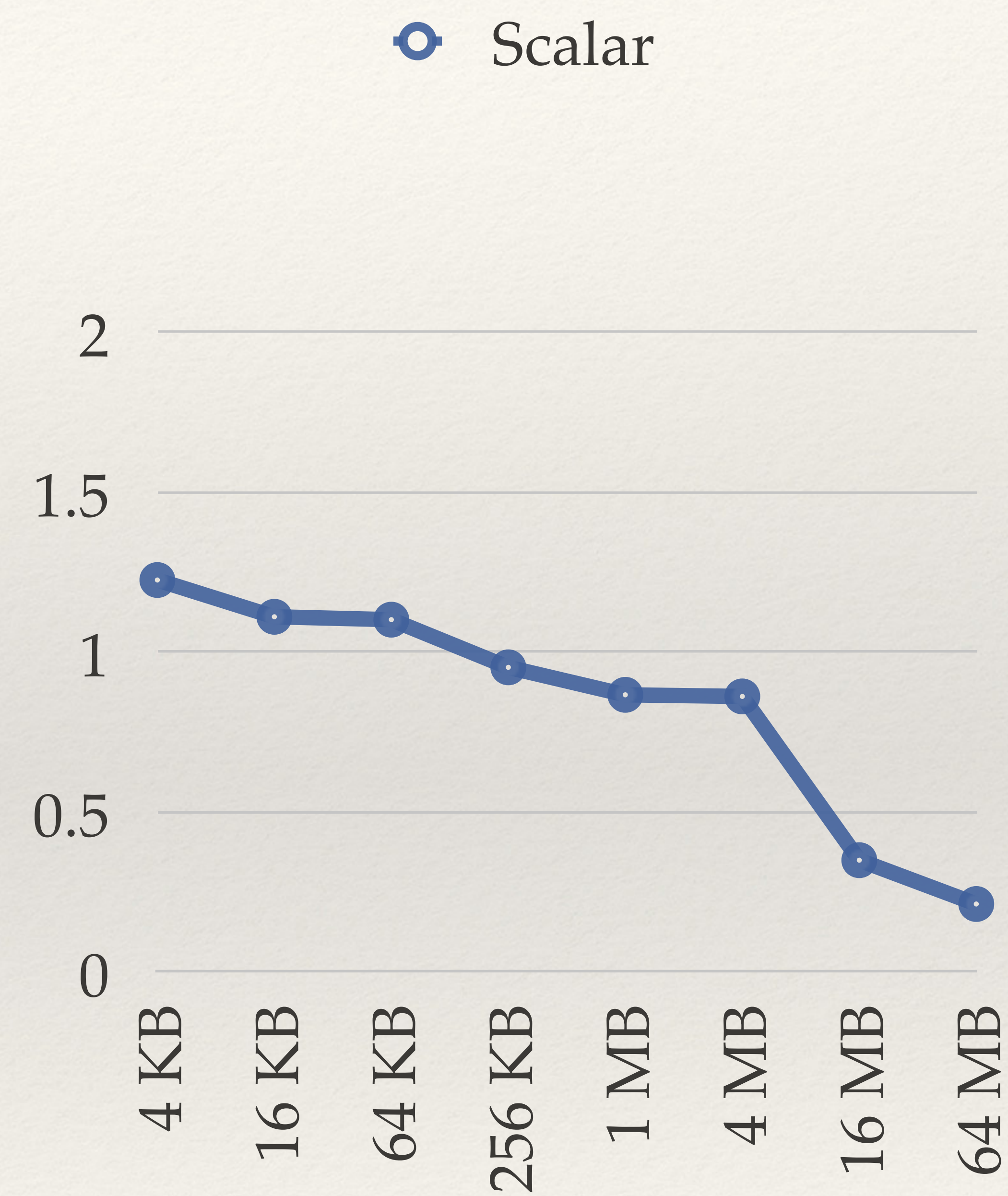
- Vectorized hash tables
  - ***Vertical*** vectorization
    - W ***input*** keys at a time
    - 1 ***table*** keys per input key
    - ***Gather*** buckets
  - Probing *(linear probing)*
    - ***Store*** matching lanes
    - ***Replace*** finished lanes
    - ***Keep*** unfinished lanes
  - Building *(linear probing)*
    - ***Keep*** non-empty lanes
    - ***Detect*** scatter ***conflicts***
    - ***Scatter*** empty lanes & ***replace***
    - ***Keep*** conflicting lanes



Hash Table (Linear Probing)

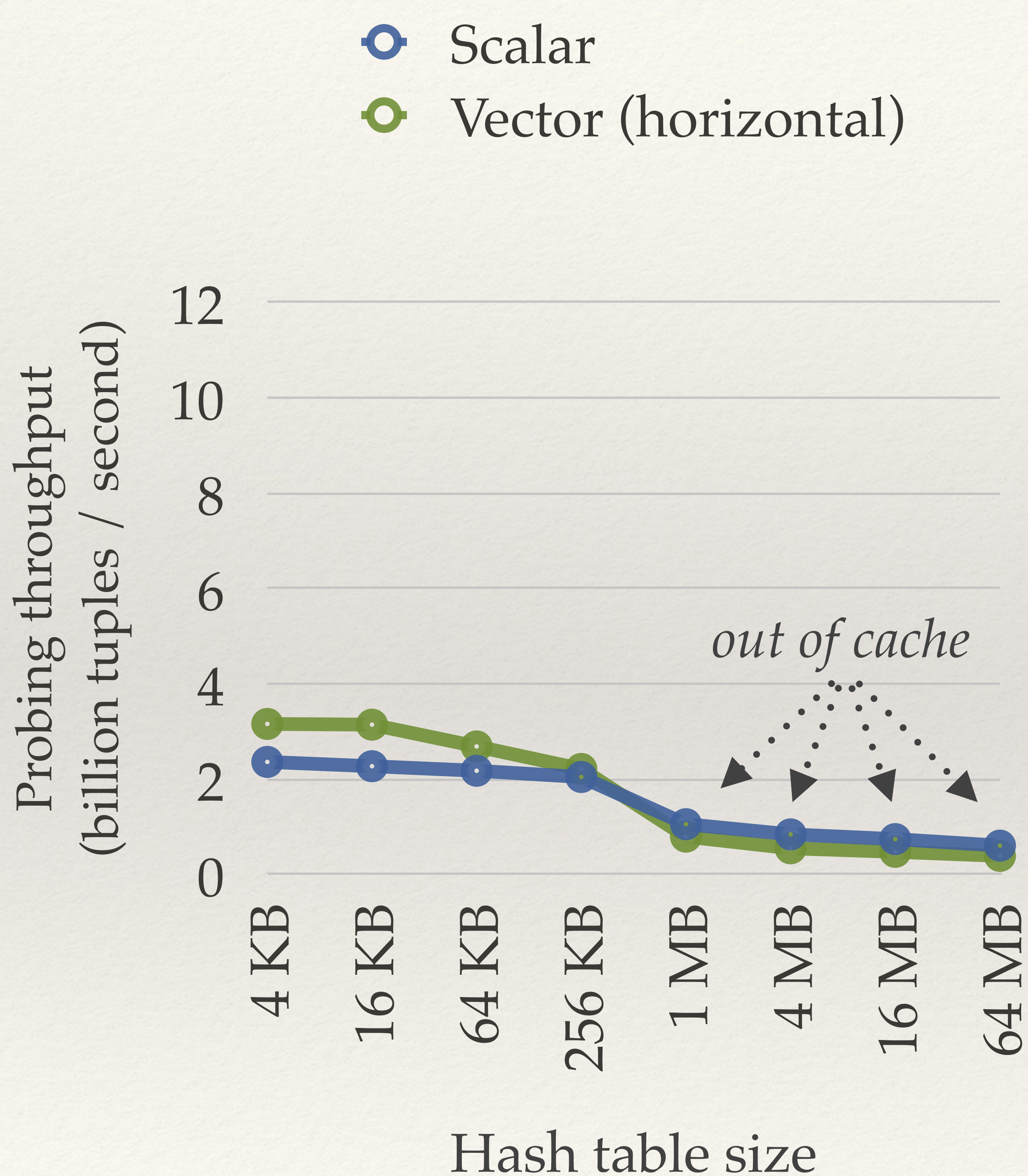


Hash table size
Xeon Phi 7120P

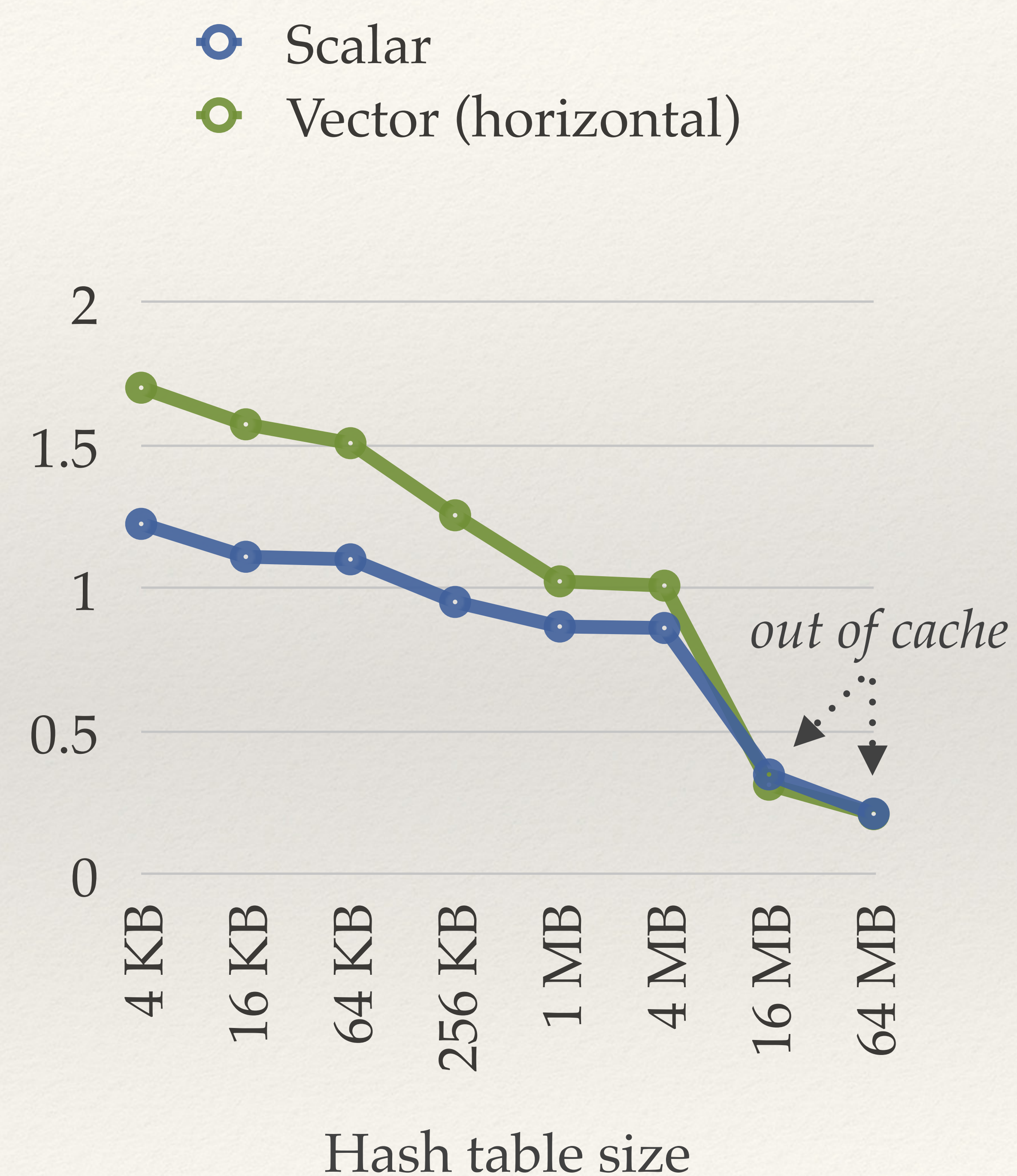


Hash table size
Xeon E3-1275v3 (**Haswell**)

Hash Table (Linear Probing)

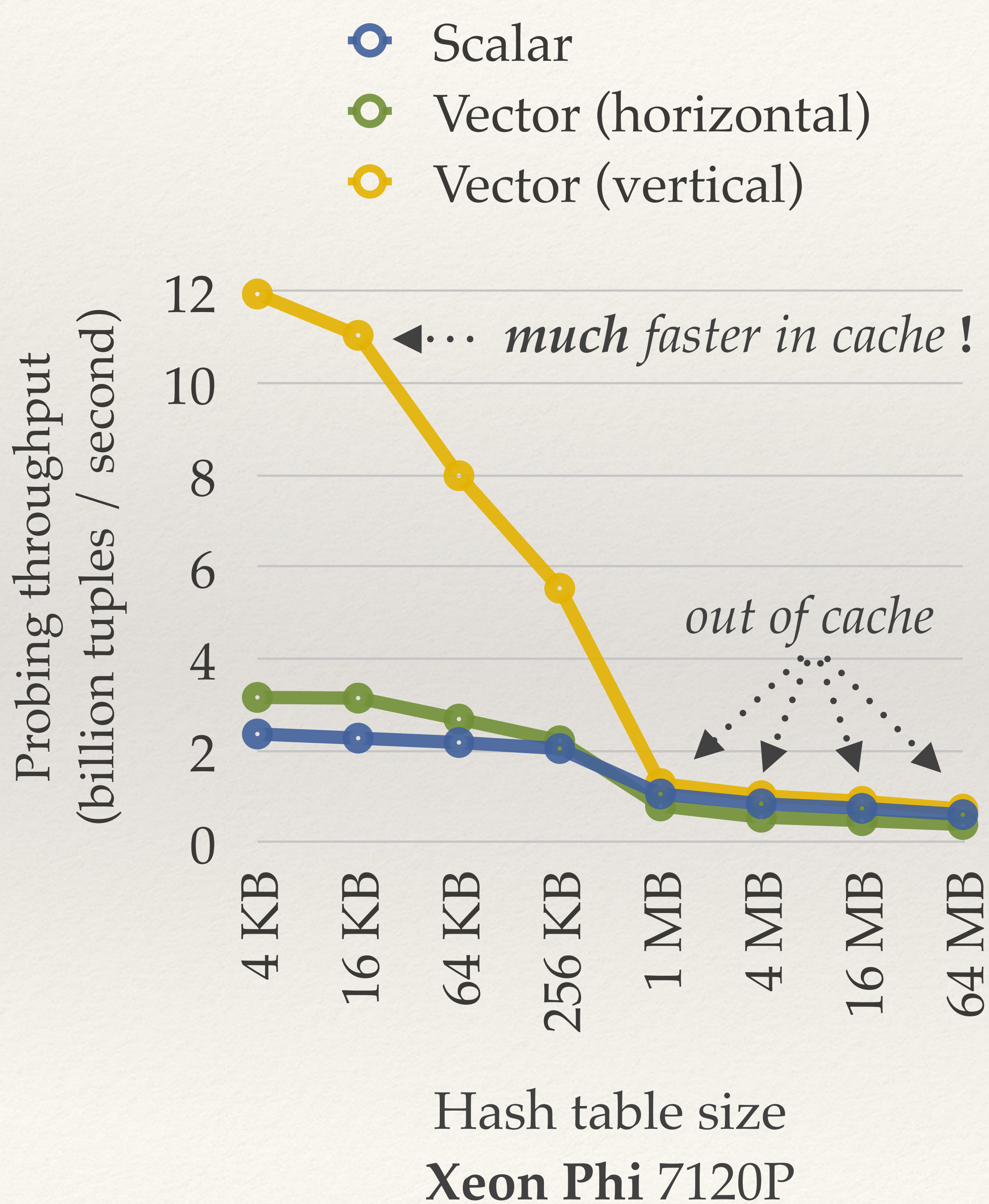


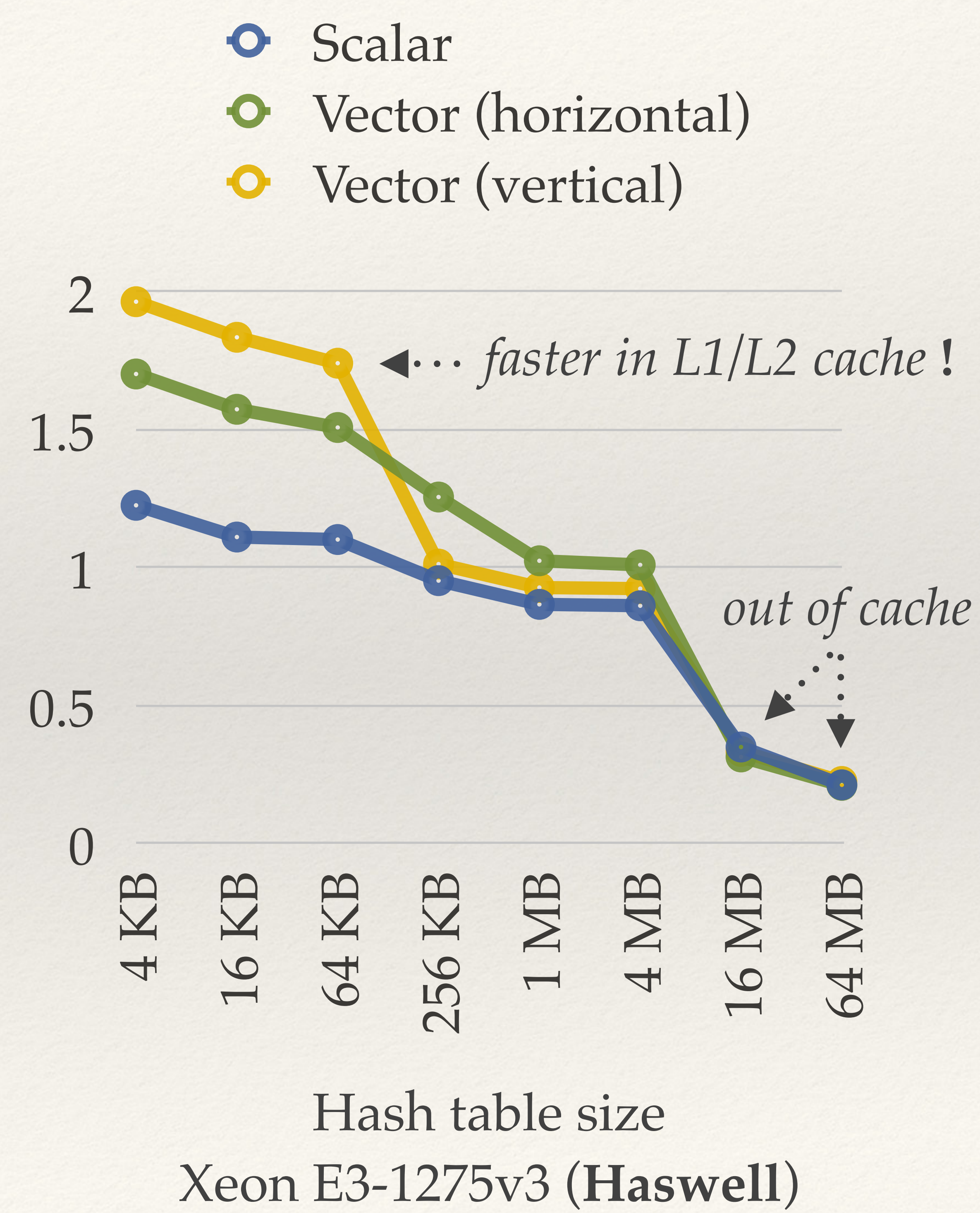
Xeon Phi 7120P



Xeon E3-1275v3 (**Haswell**)

Hash Table (Linear Probing)



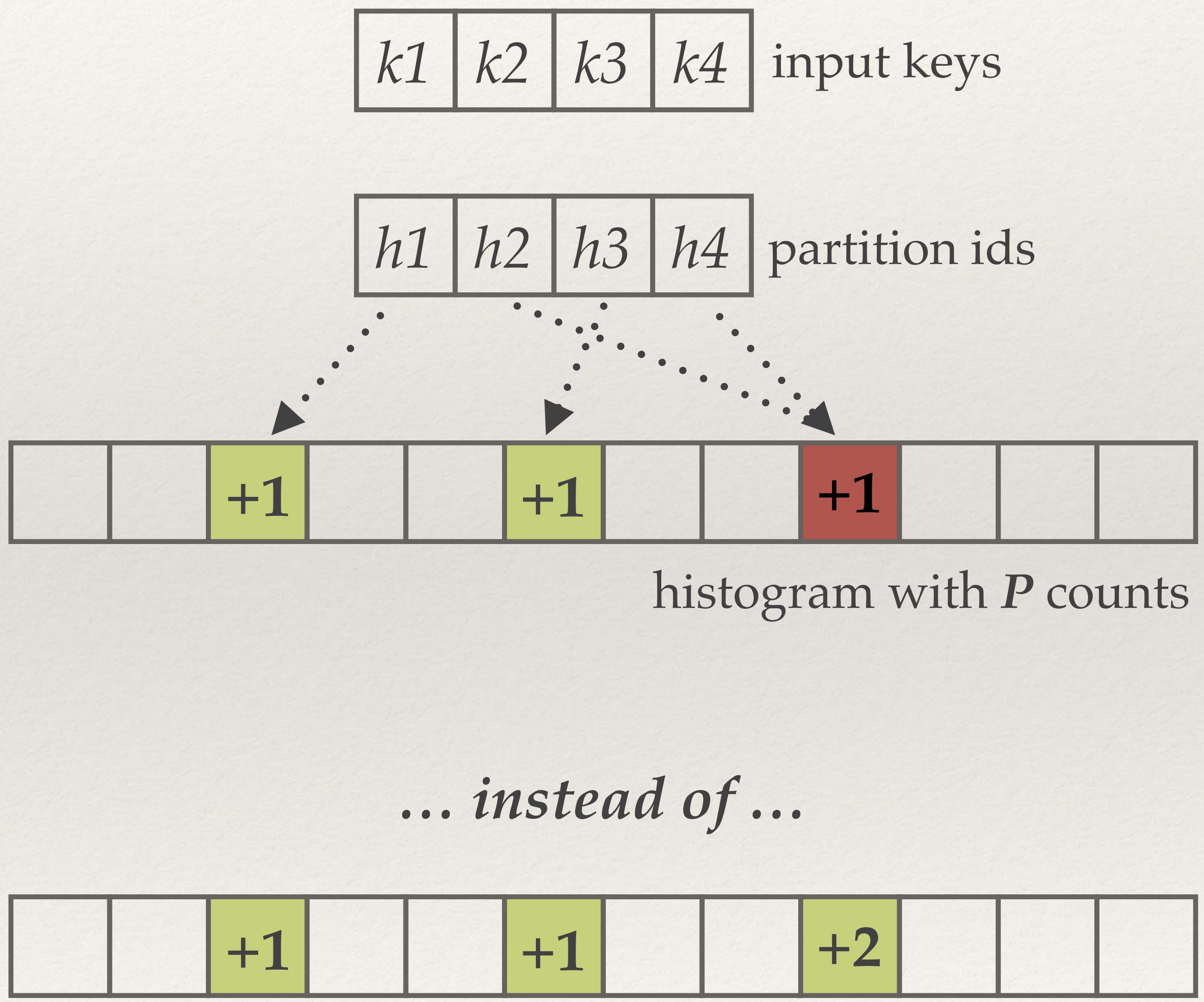


Partitioning

- Types
  - Radix
    - 2 shifts (in SIMD)
  - Hash
    - 2 multiplications (in SIMD)
  - Range
    - Range function index
    - Binary search (in the paper …)

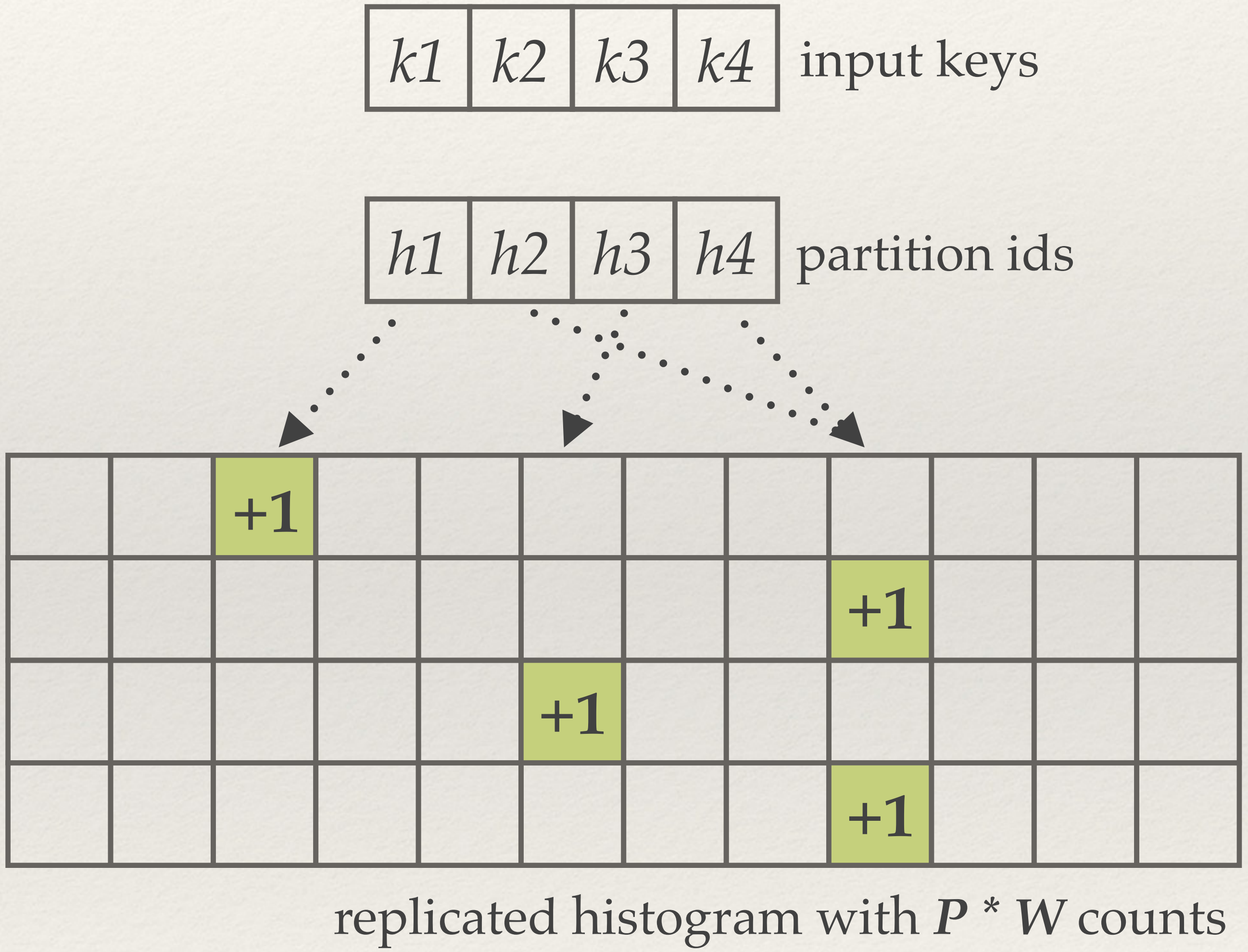
Partitioning

- Types
  - Radix
    - 2 shifts (in SIMD)
  - Hash
    - 2 multiplications (in SIMD)
  - Range
    - Range function index
    - Binary search (in the paper …)
- Histogram
  - Data parallel update
    - Gather & scatter counts
    - *Conflicts* miss counts



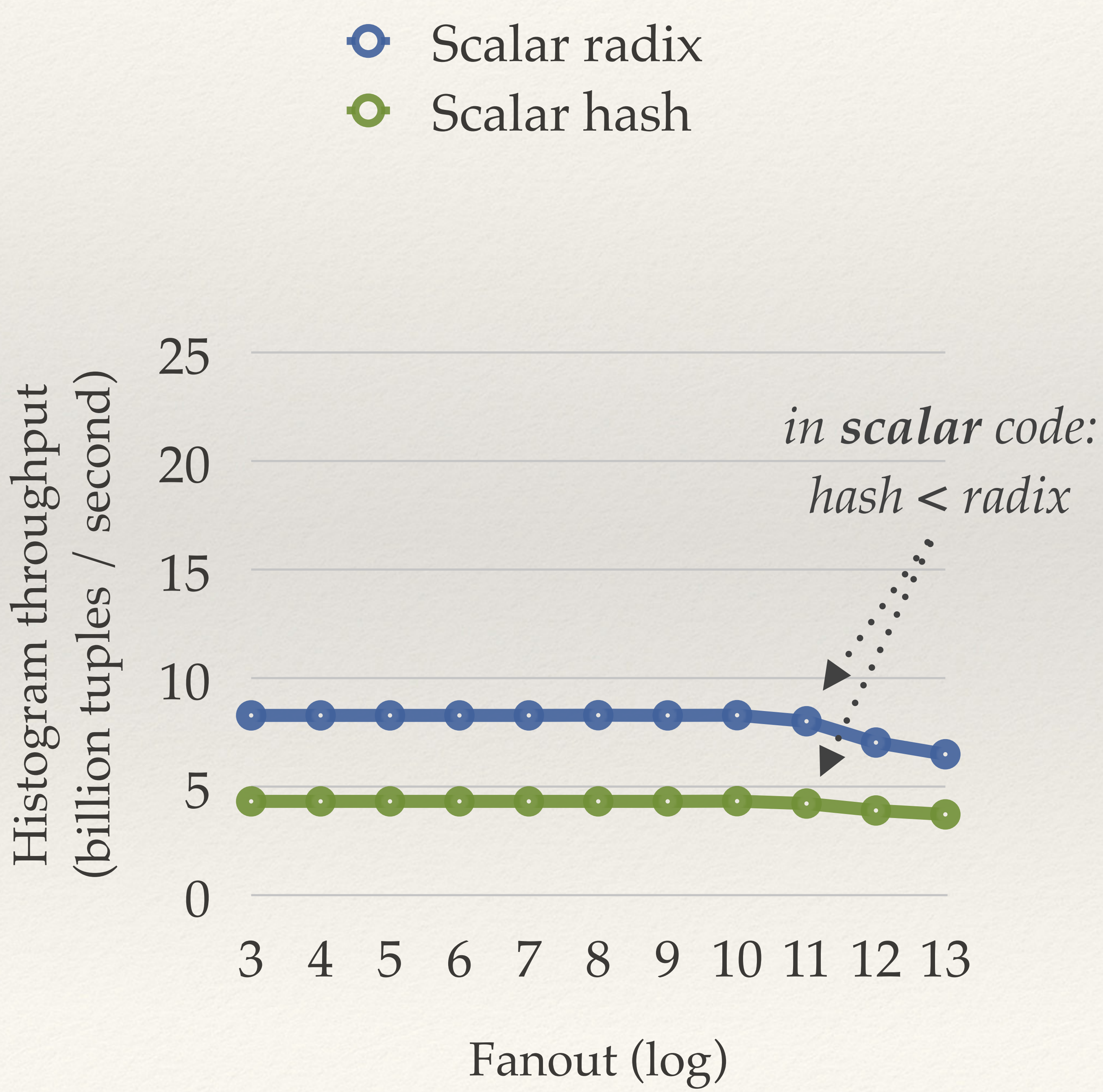
Partitioning

- Types
  - Radix
    - 2 shifts (in SIMD)
  - Hash
    - 2 multiplications (in SIMD)
  - Range
    - Range function index
    - Binary search (in the paper …)
- Histogram
  - Data parallel update
    - Gather & scatter counts
    - *Conflicts* miss counts
    - *Replicate* histogram ***W*** times
    - More solutions in the paper …



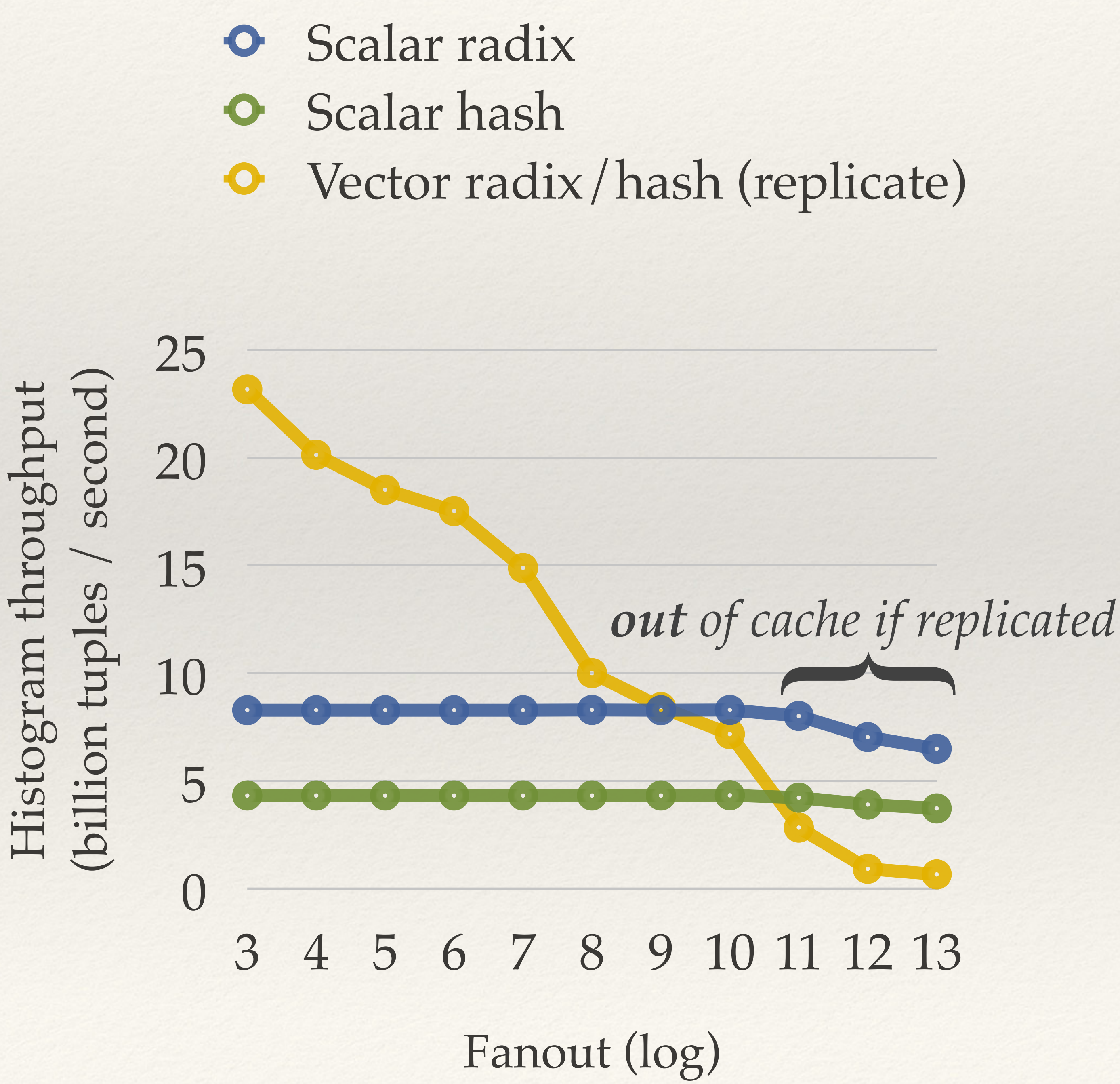
Radix & Hash Histogram

Histogram generation on **Xeon Phi**



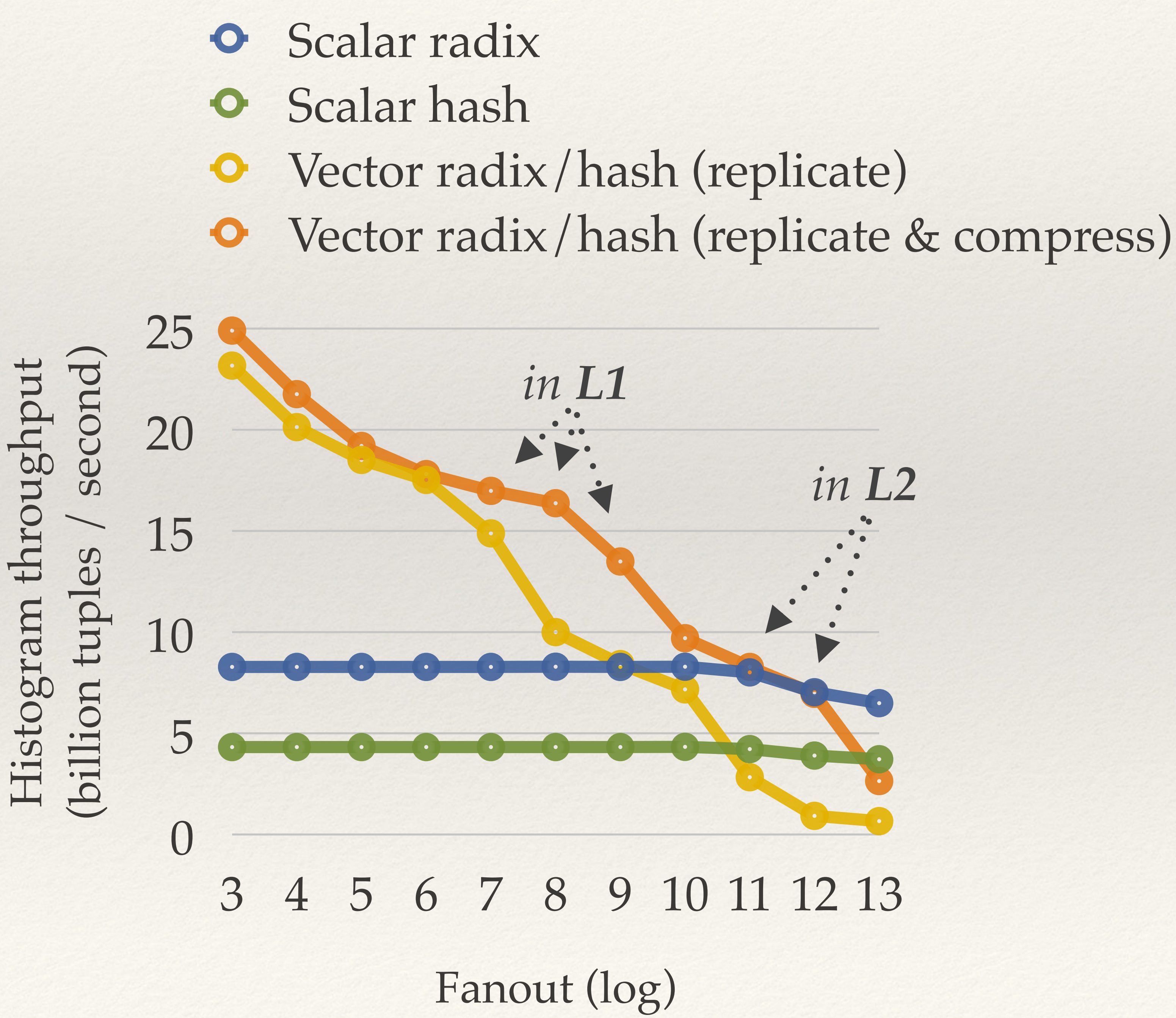
Radix & Hash Histogram

Histogram generation on **Xeon Phi**



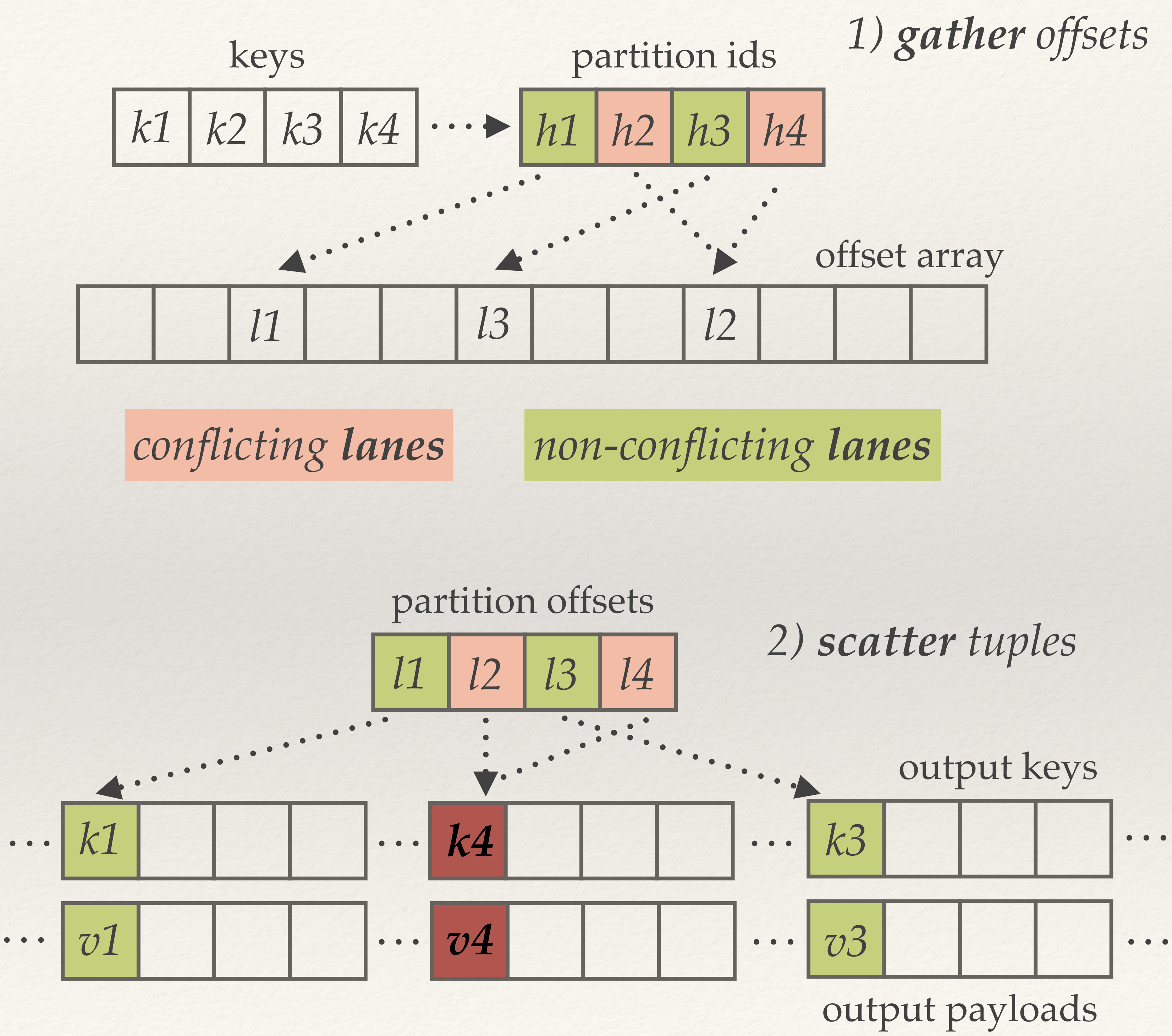
Radix & Hash Histogram

Histogram generation on **Xeon Phi**



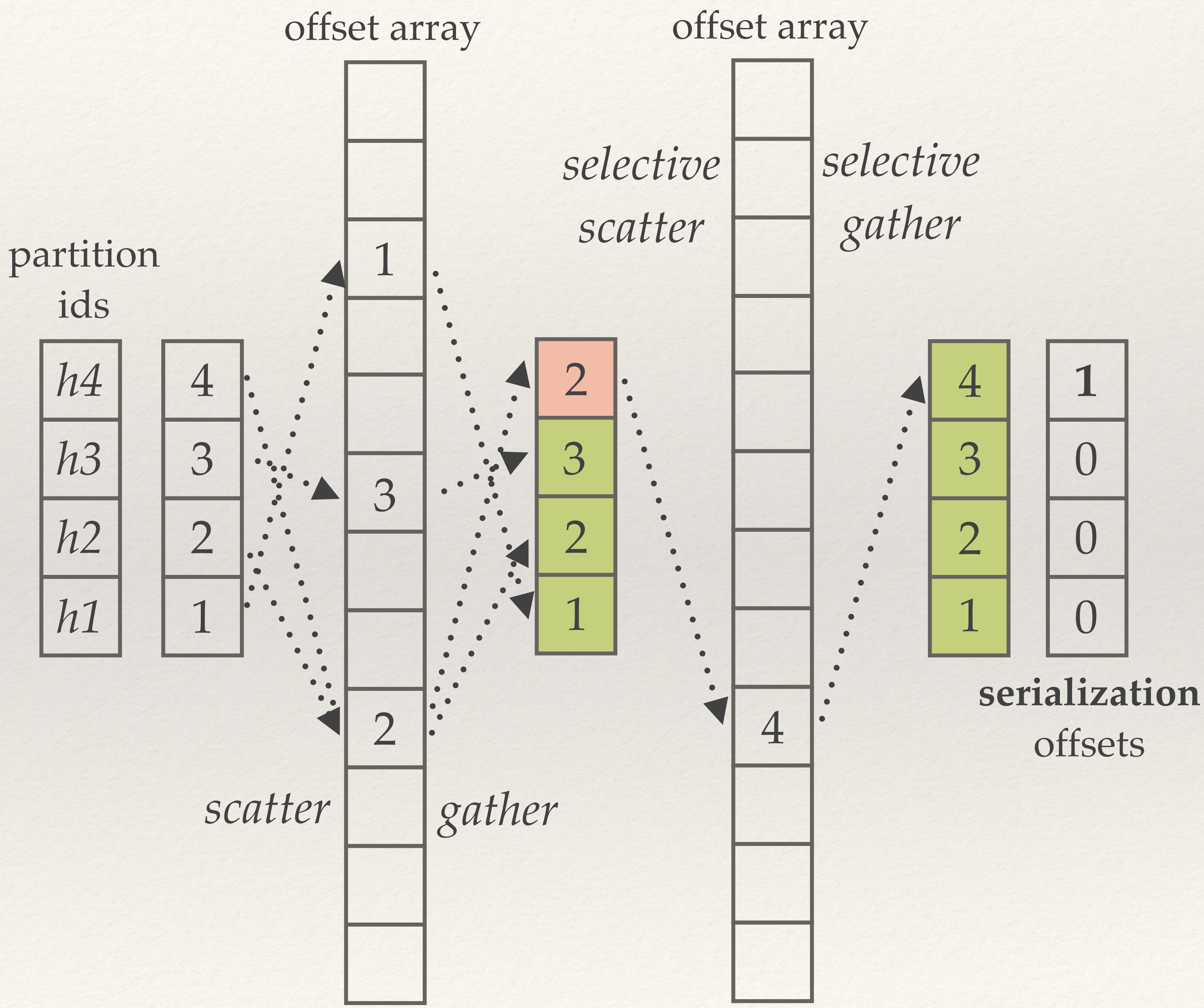
Partitioning

- Shuffling
  - Update the offsets
    - Gather & scatter counts
    - ***Conflicts*** miss counts
  - Tuple transfer
    - Scatter directly to ***output***
    - ***Conflicts*** overwrite tuples



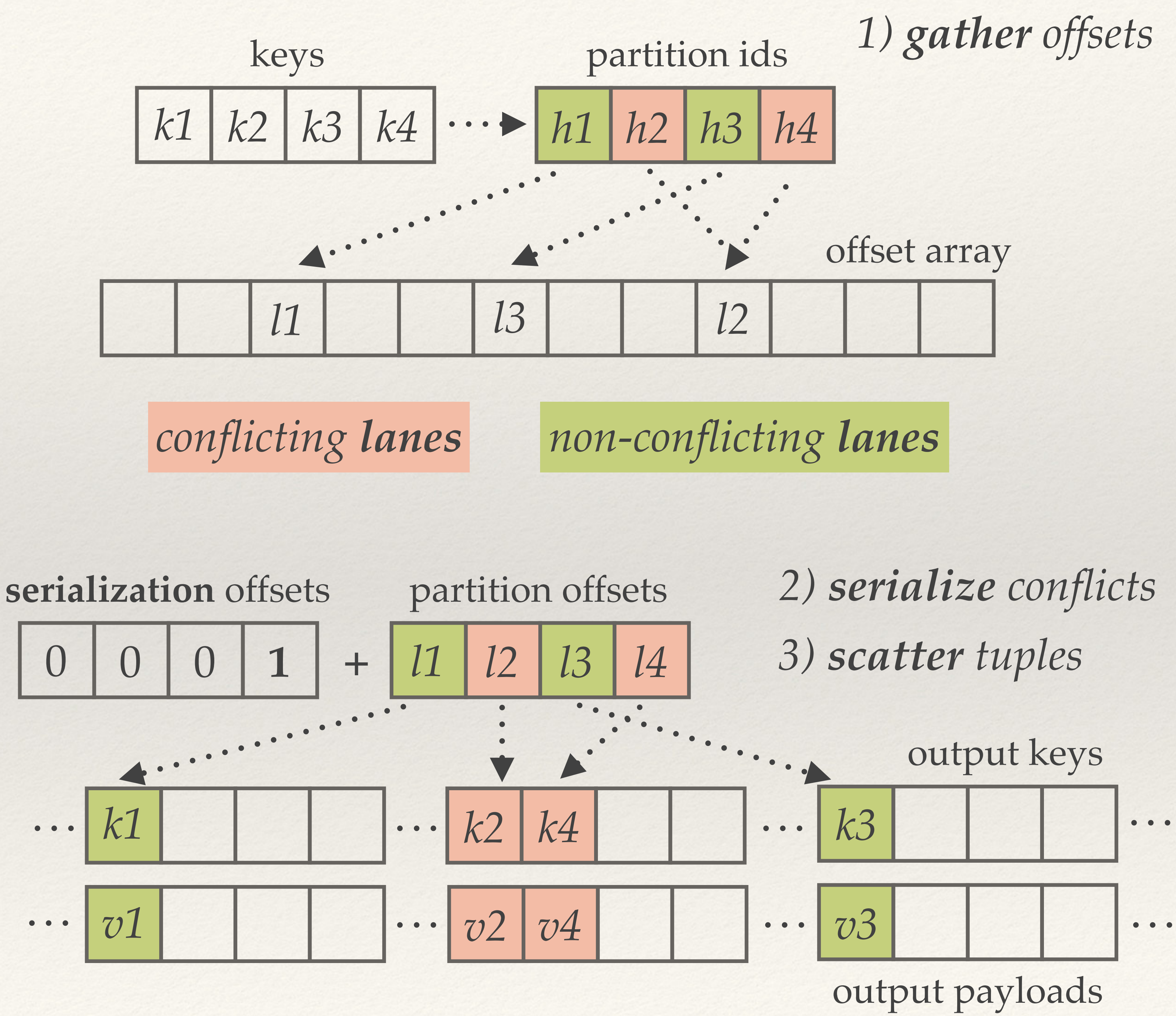
Partitioning

- Shuffling
  - Update the offsets
    - Gather & scatter counts
    - *Conflicts* miss counts
  - Tuple transfer
    - Scatter directly to output
    - *Conflicts* overwrite tuples
    - *Serialize* conflicts



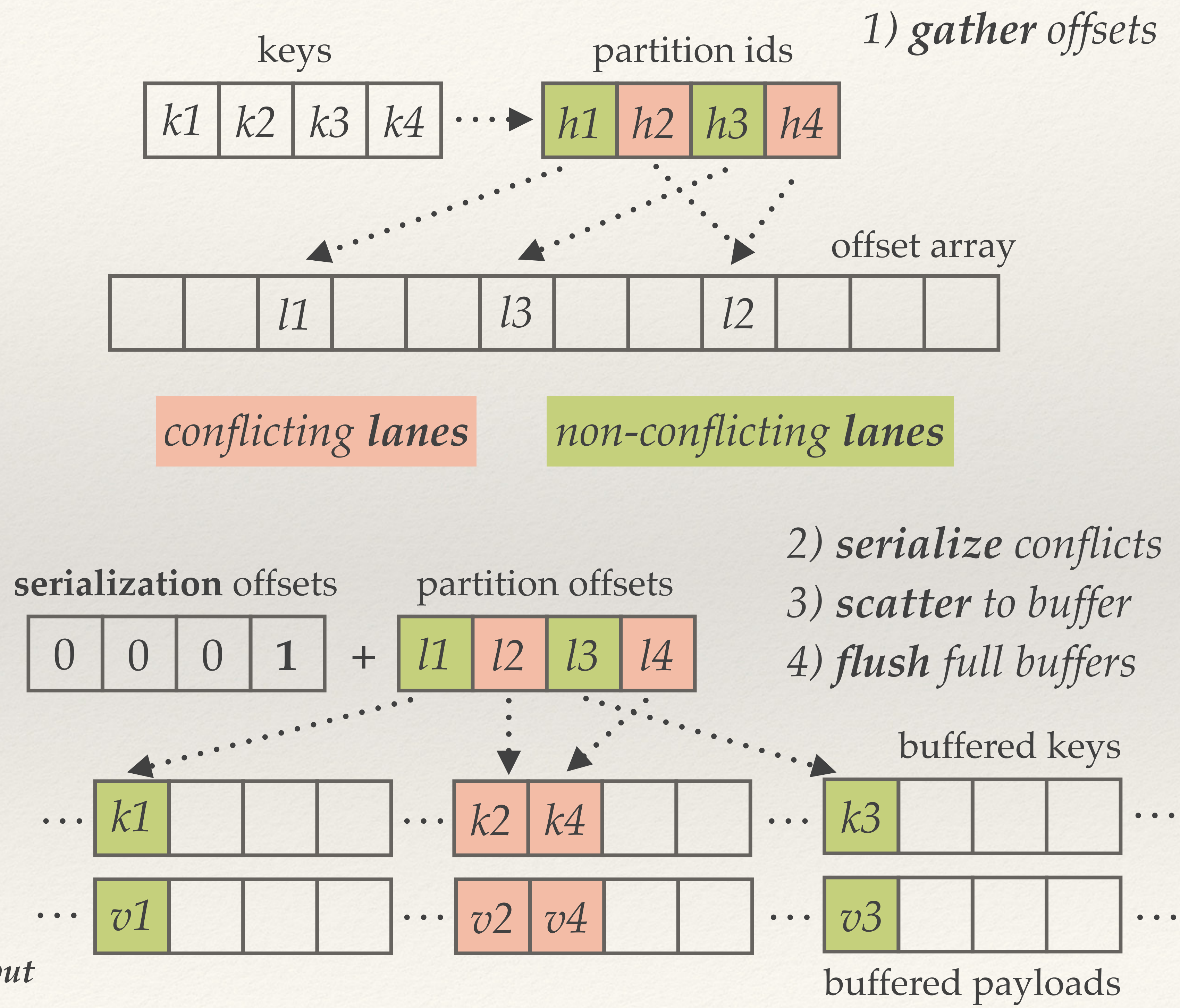
Partitioning

- Shuffling
  - Update the offsets
    - Gather & scatter counts
    - ***Conflicts*** miss counts
  - Tuple transfer
    - Scatter directly to output
    - ***Conflicts*** overwrite tuples
    - ***Serialize*** conflicts
    - More in the paper …



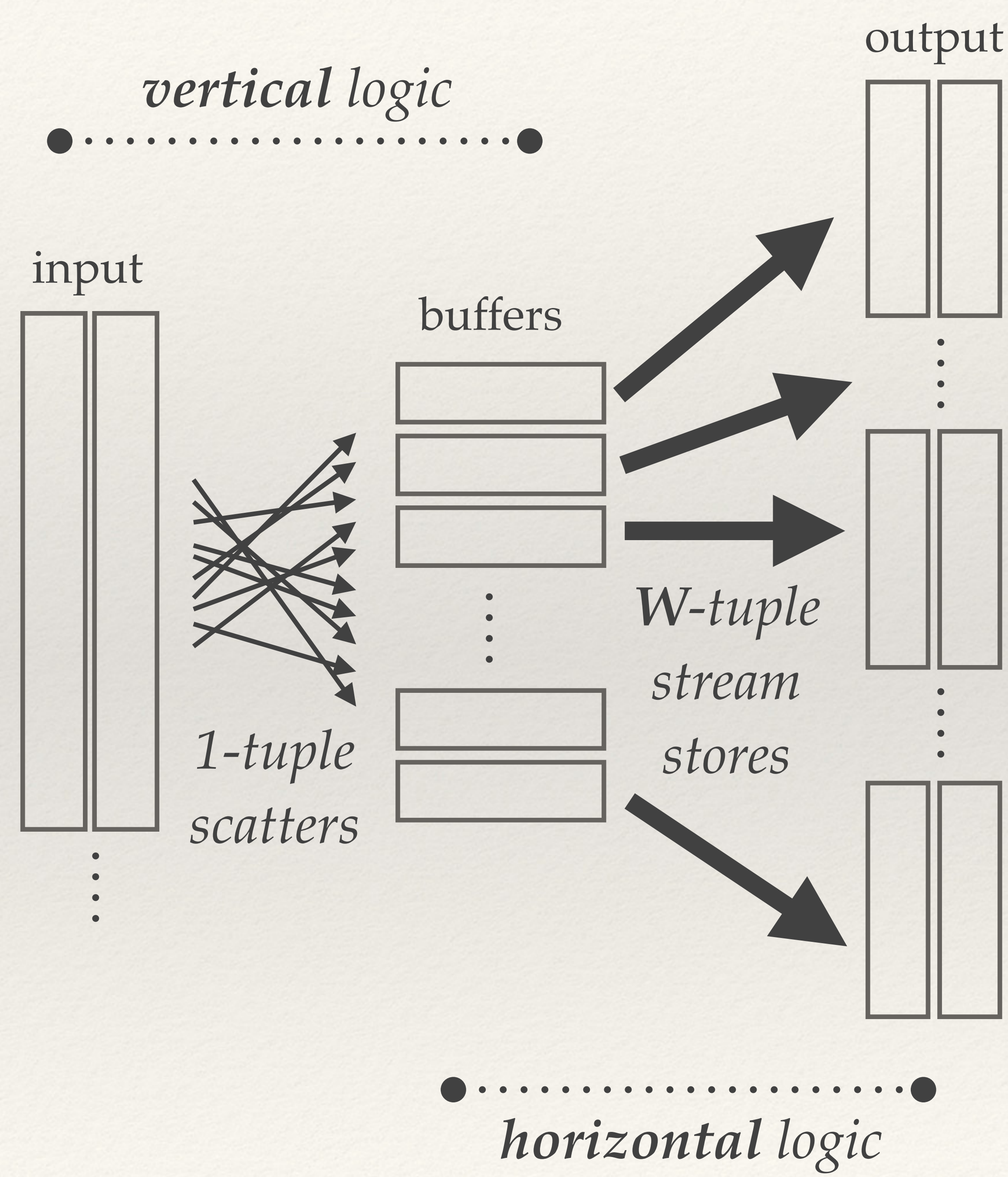
Partitioning

- Shuffling
  - Update the offsets
    - Gather & scatter counts
    - ***Conflicts*** miss counts
  - Tuple transfer
    - Scatter directly to output
    - ***Conflicts*** overwrite tuples
    - ***Serialize*** conflicts
    - More in the paper …
  - Buffering
    - When input >> cache
    - Handle ***conflicts***
    - Scatter tuples to ***buffers***
    - Buffers are ***cache***-resident
    - Stream ***full*** buffers to ***output***



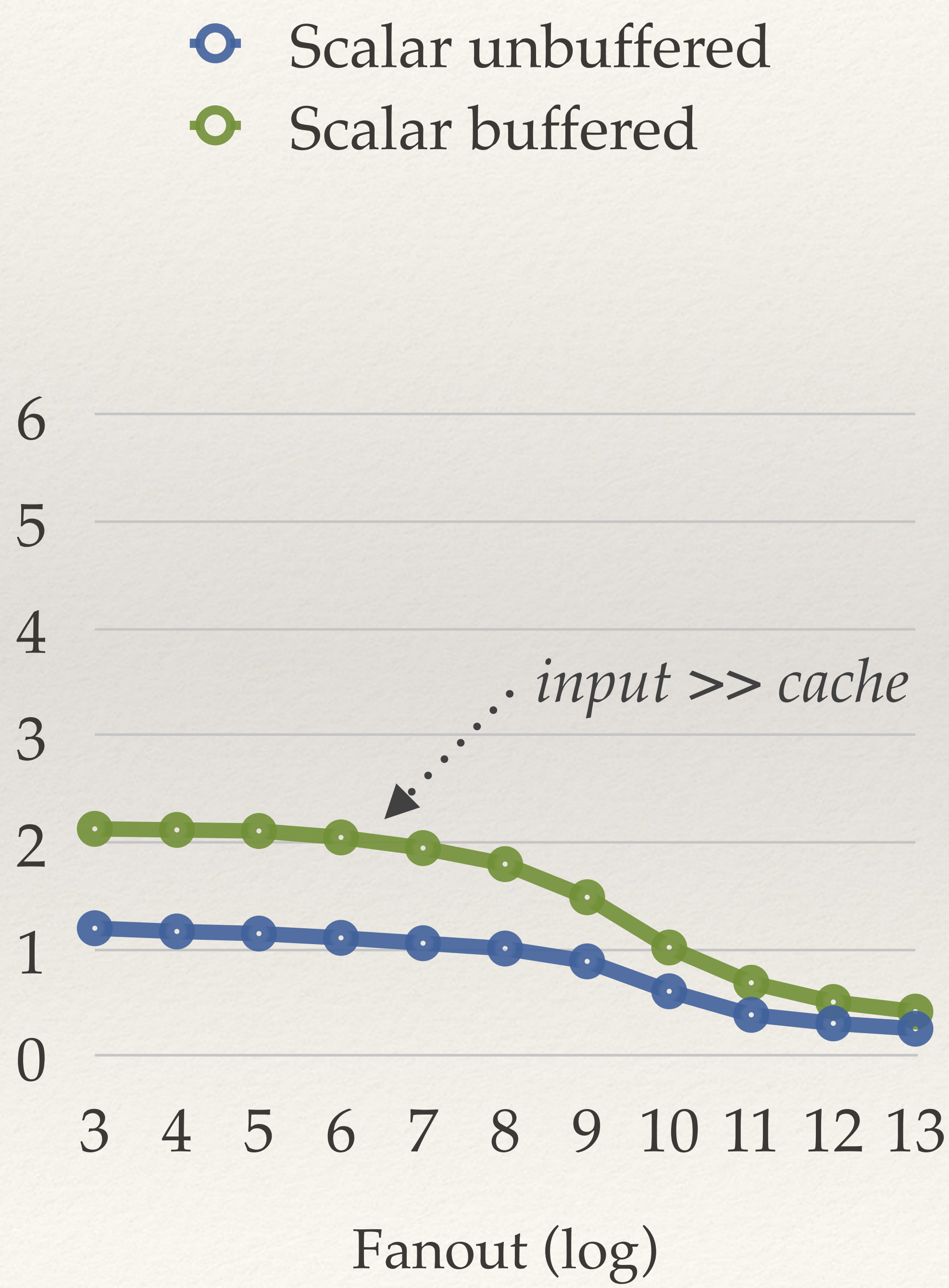
Partitioning

- Shuffling
  - Update the offsets
    - Gather & scatter counts
    - ***Conflicts*** miss counts
  - Tuple transfer
    - Scatter directly to output
    - ***Conflicts*** overwrite tuples
    - ***Serialize*** conflicts
    - More in the paper …
  - Buffering
    - When input >> cache
    - Handle ***conflicts***
    - Scatter tuples to ***buffers***
    - Buffers are ***cache***-resident
    - Stream ***full*** buffers to ***output***

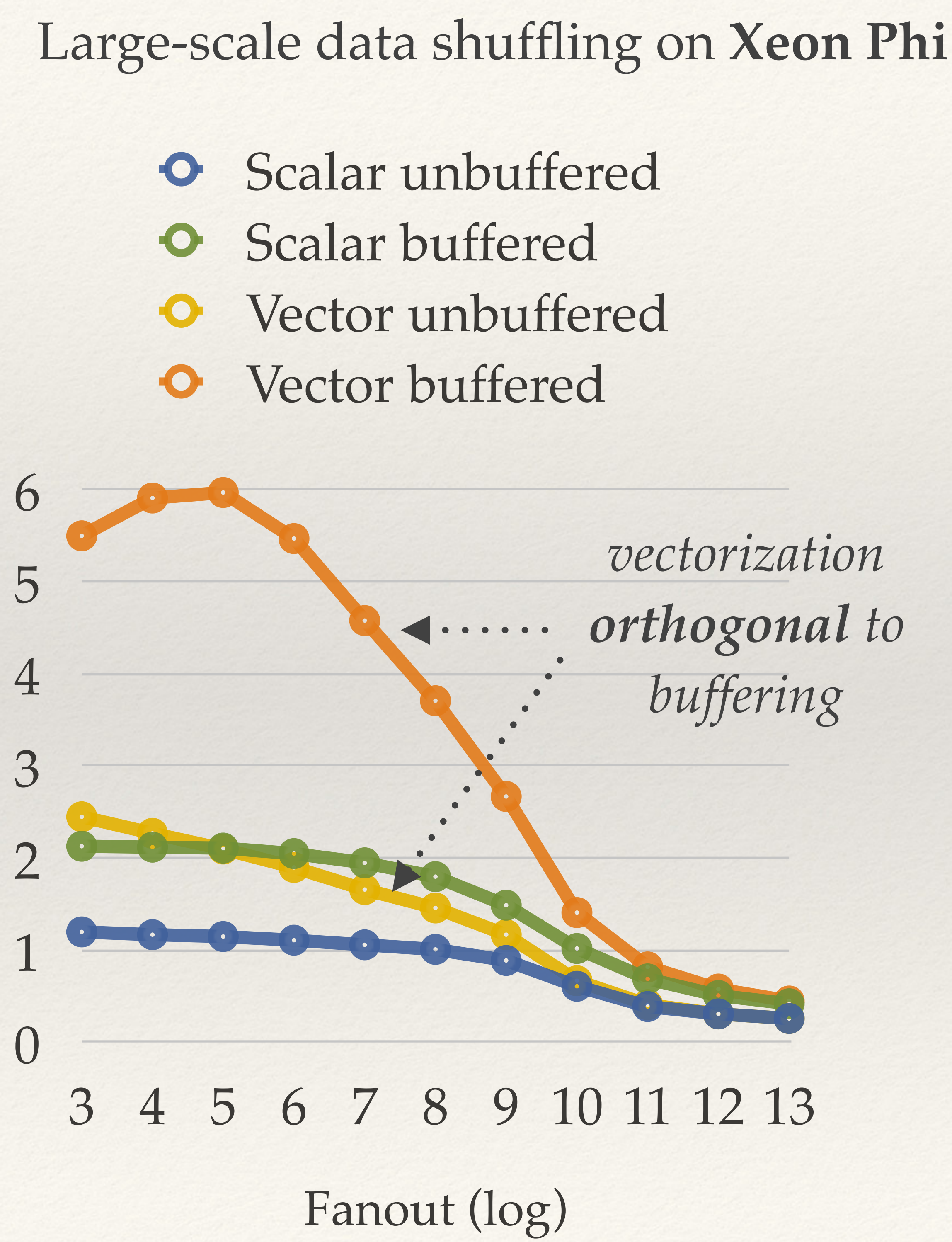


Buffered & Unbuffered Partitioning

Large-scale data shuffling on **Xeon Phi**



Buffered & Unbuffered Partitioning



Sorting & Hash Join Algorithms

- Least-significant-bit (LSB) radix-sort
  - ***Stable*** radix partitioning passes
    - ***Fully*** vectorized

Sorting & Hash Join Algorithms

- Least-significant-bit (LSB) radix-sort
  - ***Stable*** radix partitioning passes
    - ***Fully*** vectorized
- Hash join
  - ***No*** partitioning
    - Build 1 ***shared*** hash table using ***atomics***
    - ***Partially*** vectorized

Sorting & Hash Join Algorithms

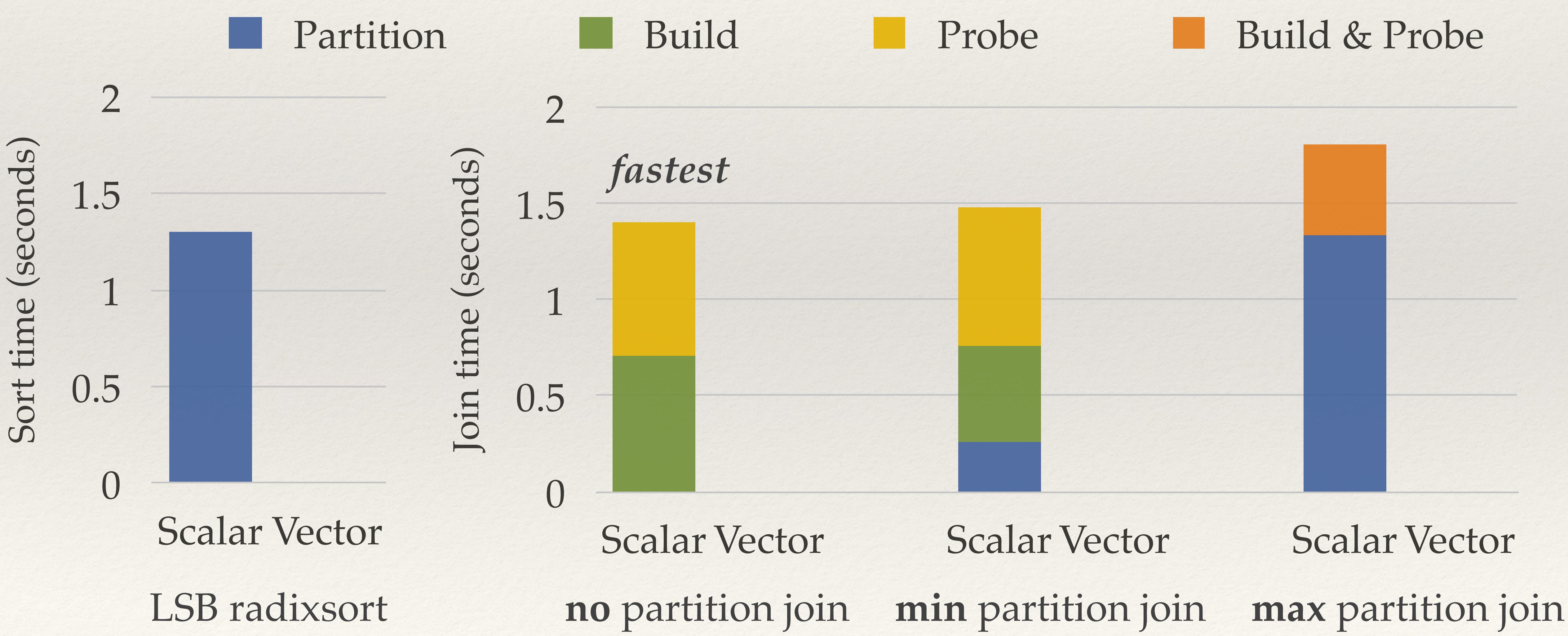
- Least-significant-bit (LSB) radix-sort
  - ***Stable*** radix partitioning passes
    - ***Fully*** vectorized
- Hash join
  - ***No*** partitioning
    - Build 1 ***shared*** hash table using ***atomics***
    - ***Partially*** vectorized
  - ***Min*** partitioning
    - Partition ***building*** table
    - Build 1 hash table per thread
    - ***Fully*** vectorized

Sorting & Hash Join Algorithms

- Least-significant-bit (LSB) radix-sort
  - ***Stable*** radix partitioning passes
    - ***Fully*** vectorized
- Hash join
  - ***No*** partitioning
    - Build 1 ***shared*** hash table using ***atomics***
    - ***Partially*** vectorized
  - ***Min*** partitioning
    - Partition ***building*** table
    - Build 1 hash table per thread
    - ***Fully*** vectorized
  - ***Max*** partitioning
    - Partition ***both*** tables repeatedly
    - Build & probe ***cache-resident*** hash tables
    - ***Fully*** vectorized

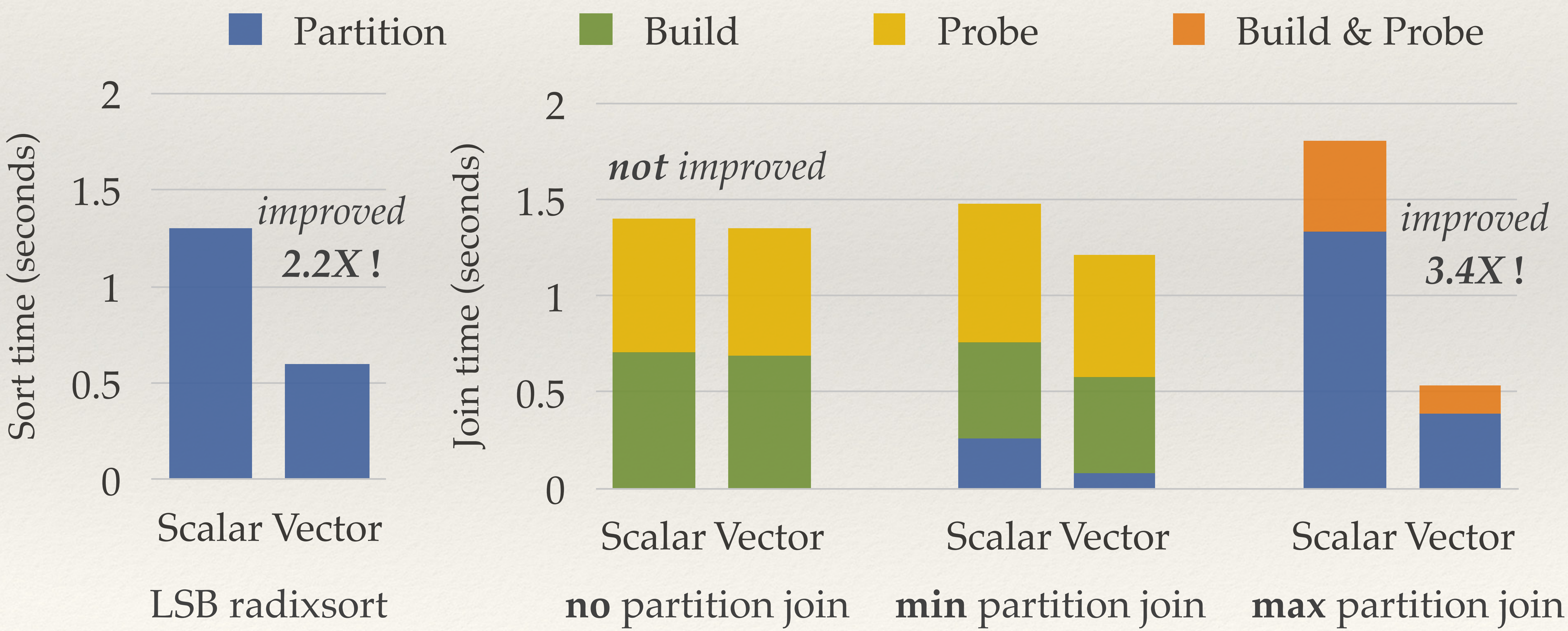
Hash Joins

sort 400 million tuples &
join 200 & 200 million tuples
32-bit keys & payloads
on **Xeon Phi**



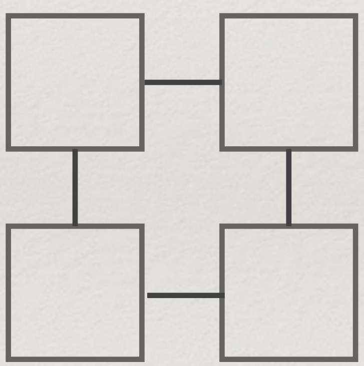
Hash Joins

sort 400 million tuples &
join 200 & 200 million tuples
32-bit keys & payloads
on **Xeon Phi**

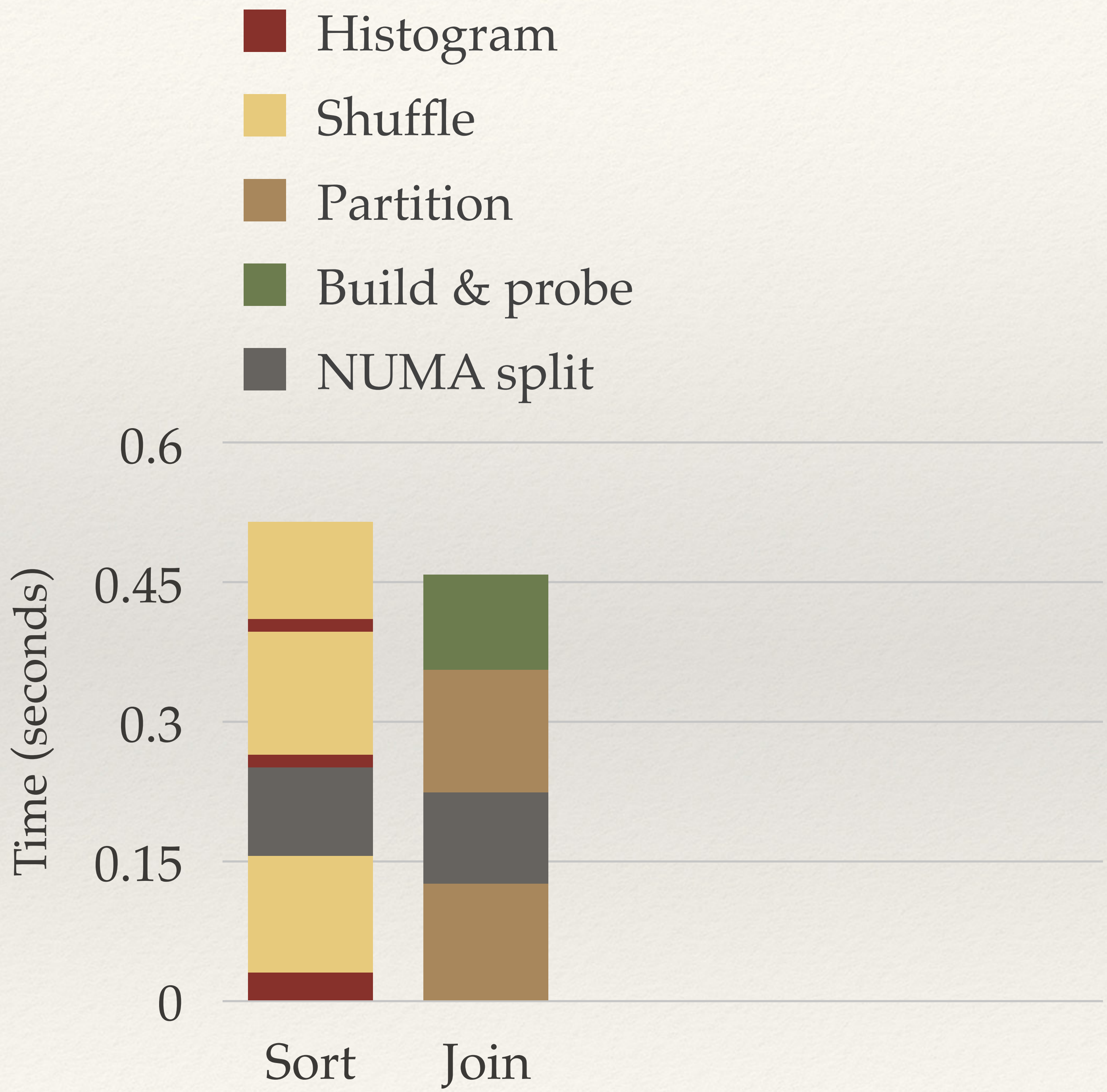


Power Efficiency

4 CPUs
(Xeon E5-4620)
32 **Sandy Bridge** cores
300 Watts TDP

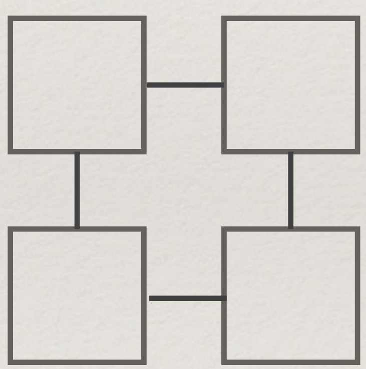


2.2 GHz
40 GB/s

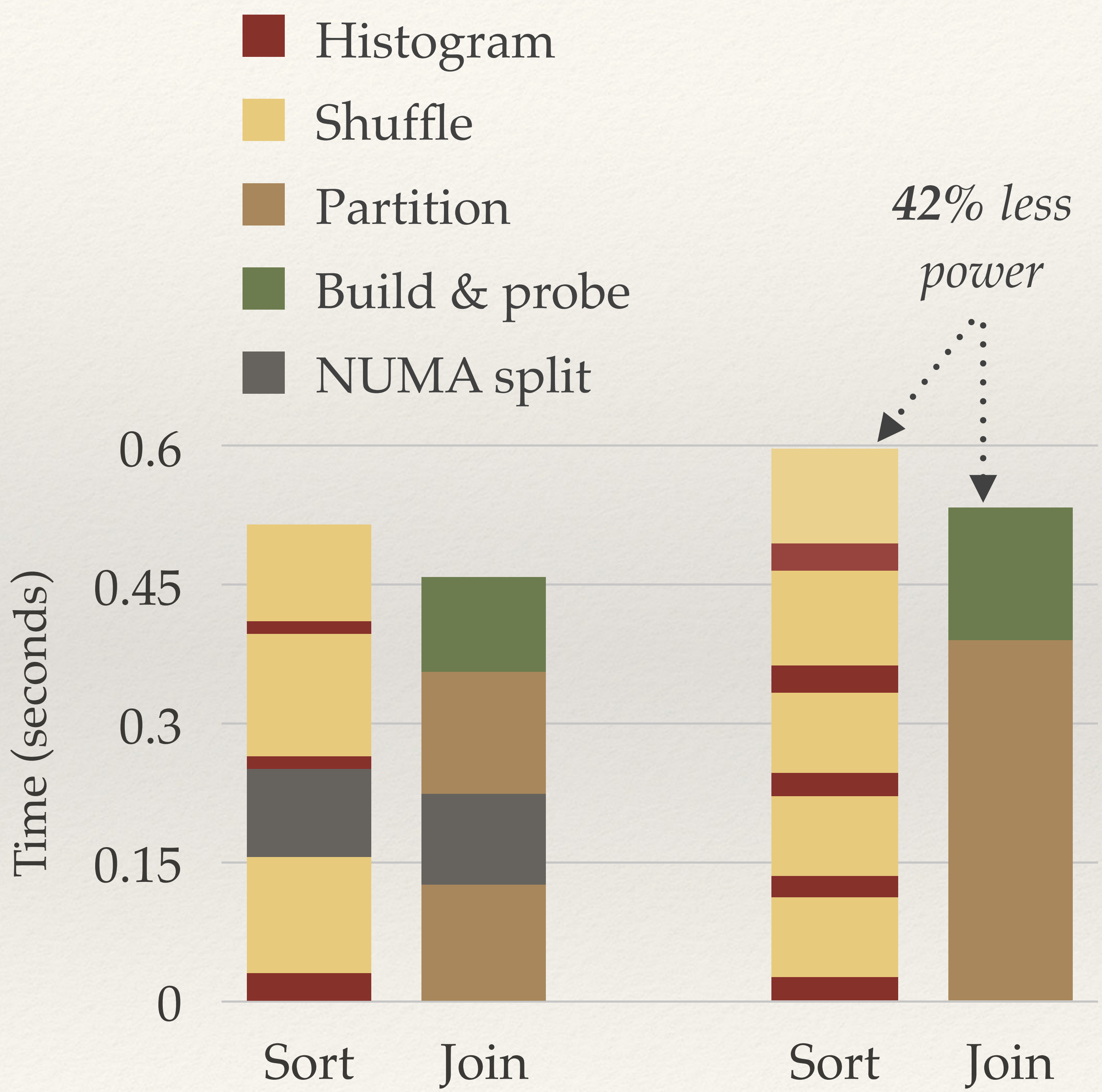


Power Efficiency

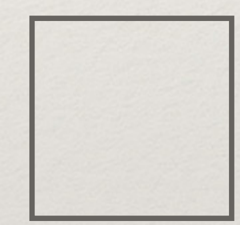
4 CPUs
(Xeon E5-4620)
32 **Sandy Bridge** cores
300 Watts TDP



2.2 GHz
40 GB/s *



1 co-processor
(Xeon Phi 7120P)
61 **P54C** cores
520 Watts TDP



1.238 GHz
80 GB/s *

* **comparable** speed even if equalized (in the paper …)

Conclusions

- ***Full*** vectorization
  - $O(f(n))$ scalar —> $O(f(n)/W)$ vector operations
  - Good vectorization principles improve ***performance***
    - Define & ***reuse*** fundamental operations
    - e.g. ***vertical*** vectorization, maximize lane ***utilization***, …

Conclusions

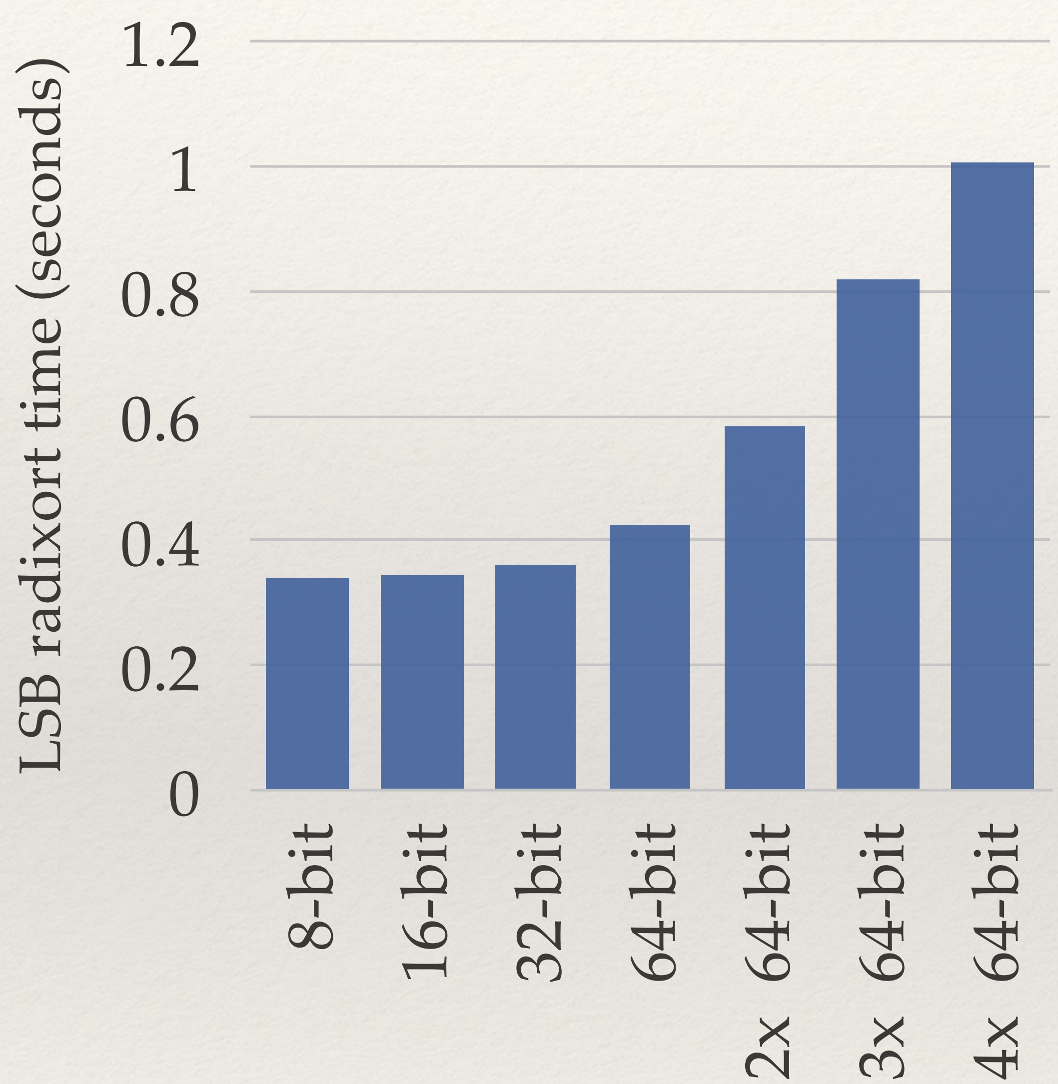
- ***Full*** vectorization
  - ***O(f(n))*** scalar —> ***O(f(n)/W)*** vector operations
  - Good vectorization principles improve ***performance***
    - Define & ***reuse*** fundamental operations
    - e.g. ***vertical*** vectorization, maximize lane ***utilization***, …
- Impact on software design
  - Vectorization favors cache-conscious algorithms
    - e.g. ***partitioned*** hash join >> non-partitioned hash join if vectorized
  - Vectorization is ***orthogonal*** to other optimizations
    - e.g. ***both*** unbuffered & ***buffered*** partitioning get vectorization speedup

Conclusions

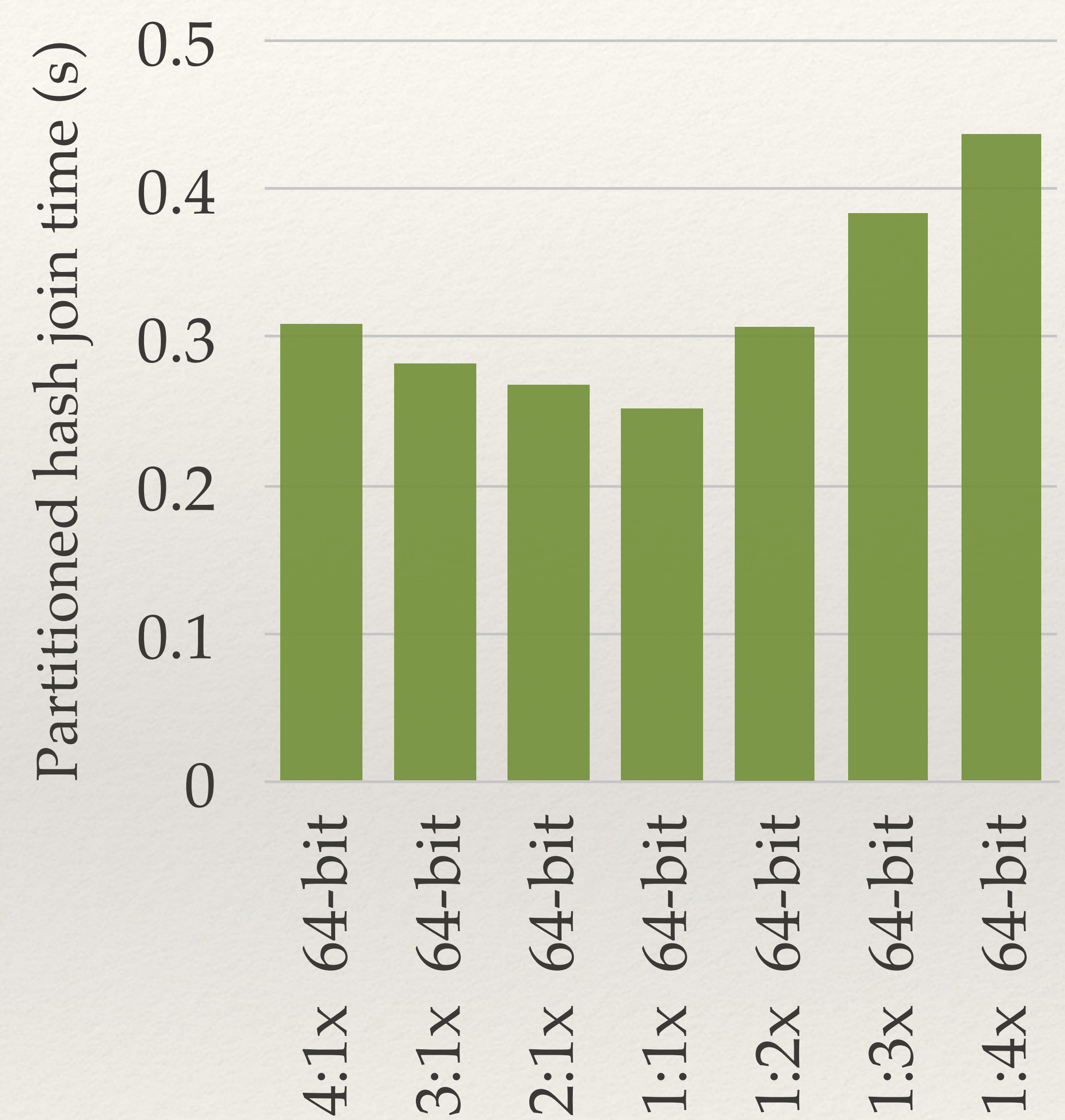
- ***Full*** vectorization
  - ***O(f(n))*** scalar —> ***O(f(n)/W)*** vector operations
  - Good vectorization principles improve ***performance***
    - Define & ***reuse*** fundamental operations
    - e.g. ***vertical*** vectorization, maximize lane ***utilization***, …
- Impact on software design
  - Vectorization favors cache-conscious algorithms
    - e.g. ***partitioned*** hash join >> non-partitioned hash join if vectorized
  - Vectorization is ***orthogonal*** to other optimizations
    - e.g. ***both*** unbuffered & ***buffered*** partitioning get vectorization speedup
- Impact on hardware design
  - ***Simple*** cores almost as fast as ***complex*** cores (for OLAP)
    - 61 simple ***P54C*** cores ~ 32 complex ***Sandy Bridge*** cores
  - Improved ***power*** efficiency for analytical databases

Questions

Join & Sort with Early Materialized Payloads



200 million 32-bit keys & X payloads on **Xeon Phi**



10 & 100 million 32-bit keys & X payloads on **Xeon Phi**